Algorithms and Programming Tools for Image Processing on the MPP: #2

Report for the Period
March 1986 to August 1985

Anthony P. Reeves
School of Electrical Engineering
Cornell University
Ithaca, New York 14853

Work Supported by NASA Grant NAG 5-403

Algorithms and Programming Tools for Image Processing on the MPP: #2

Report for the Period
March 1986 to August 1985

Anthony P. Reeves
School of Electrical Engineering
Cornell University
Ithaca, New York 14853

Work Supported by NASA Grant NAG 5-403

Summary

The work reported here was conducted by Maria Gutierrez and Marc Willebeek-LeMair, who are graduate students at Cornell University, and myself. The work for this period iof the grant falls into two main categories: algorithms for the MPP and performance analysis of data manipulations for the MPP and related architectures. Maria has developed a number of novel algorithms for image warping and pyramid image filtering. Marc has investigated techniques for the parallel processing of a large number of independent irregular shaped regions on the MPP. In addition some new utilities for dealing with very long vectors and for sorting have been developed. Documentation pages for the new algorithms which are available for distribution are given in Appendix A. Not all algorithms have been made to work on the MPP. The effort in the final period of the grant will concentrate on the MPP implementation.

The performance of the MPP for a number of basic data manipulations has been determined. From these results it is possible to predict the efficiency of the MPP for a number of algorithms and applications. Some of these results have been published [1, 2, 3] and these papers are included as Appendices C, D and E. The Parallel Pascal development system, which is a portable programming environment for the MPP, has been improved and better documentation including a tutorial has been written. This environment, allows programs for the MPP to be developed on any conventional computer system; it consists of a set of system programs and a library of general purpose Parallel Pascal functions. The new tutorial is included as Appendix B.

During this report period Maria, Marc and myself have visited the NASA Goddard Space Flight Center. Algorithms have been tested on the MPP and a presentation on the development system was made to the MPP users group. We have distributed the UNIX version of the Parallel Pascal System to number of new sites.

Some of the highlights of the results of this research are listed below.

Image Processing Algorithms

Algorithms for image warping are described in Appendix A. The two main functions are nearest neighbor, NNWARP, and bilinear interpolation BLWARP. Both of these functions are guided by the same heuristic technique which is very efficient for small arbitrary warps but can also deal with large image distortions.

Building on the pyramid processing primitives, which were mentioned in the previous report, Laplacian and Gaussian pyramid image filters have been implemented by the functions LAPLACIAN and GAUSSIAN respectively as outlined in Appendix A. These algorithms are used to decompose an image into a number of bandpass filtered subimages. A number of interesting efficient image analysis and image filtering algorithms have been based on this pyramid of subimages.

Local Region Processing

A new approach to the parallel processing of independent regions in parallel on the MPP is being investigated. For each region in an image a tree is created which spans the region and can be used to compute features of the region. Special provisions have been made for generating trees for non-simply connected regions. Techniques for parallel region merging have been developed. After merging two regions, a new tree is generated which covers the region in a minimum amount of time. Using these tree procedures, an image segmentation algorithm, based on the split and merge paradigm, has been implemented. An initial paper on these techniques is in preparation.

General Utilities

A general purpose sorting algorithm has been implemented; this is described in the SORT documentation page in Appendix A. Based on the bitonic sorting technique this program can sort the rows, the columns or treat the whole PE matrix as a long

vector. Any basic data type can be sorted.

There are new utilities for local mean, local median and local maximum filters; see LMEAN, LMEDIAN and LMAXIMUM in Appendix A. Also, a general purpose binary matching algorithm (COMPN) has been developed.

Performance analysis

An analysis of different data permutations and manipulations on the MPP is presented in [1] which is also included in Appendix C. This analysis expresses the cost of data manipulations in terms of elemental arithmetic operations; Boolean, integer and floating point data types are considered. Results are computed for arrays of size 128 x 128, 256 x 256, and 512 x 512. An extended version of this paper, which includes a general description of the MPP, is given in [2] which is also included as Appendix C.

There has been much recent interest in the implementation of parallel pyramid data processors. Such a processor could be maded with the MPP by augmenting the processor array with a pyramid structure of additional processing elements. A pyramid processor based on the MPP is considered in [3] which is also included as Appendix D. The results from an analysis of the proposed system indicate that, in general, there would be little advantage in having the additional pyramid hardware for implementing many of the pyramid algorithms.

References

1. A. P. Reeves and C. H. Moura, "Data Manipulations on the Massively Parallel Processor," *Proceedings of the Nineteenth Hawaii International Conference on System Sciences*, pp. 222-229 (January, 1986).

2. A. P. Reeves, "The Massively Parallel Processor: A Highly Parallel Scientific Computer," pp. 239-252 in *Data Analysis in Astronomy II*, ed. V. Di Gesu, Plenum Press (1986).

3. A. P. Reeves, "Pyramid Algorithms on Processor Arrays," pp. 195-213 in *Pyramidal Systems for Computer Visison*, ed. V. Cantoni and S. Levialdi, Academic Press (1986).

NAME

blwarp – Bilinear interpolation warping

SYNOPSIS

{$library blwarp.pl}

function blwarp(mx:pli; rp:pli;cp:pli; rf:plr; cf:plr):pli; extern rl, rh, cl, ch;

TYPES

plr = array [rl..rh,cl..ch] of btype;
pli = array [rl..rh,cl..ch] of itype;
plb = array [rl..rh,cl..ch] of boolean;
Where btype is any type and itype is an integer or subrange base type

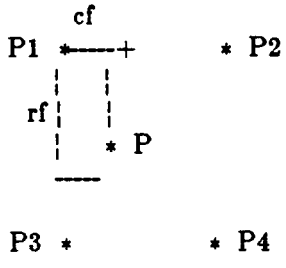
EXTERN CONSTANTS

rl = the smallest row number of the input matrix
rh = the largest row number of the input matrix
cl = the smallest column number of the input matrix
ch = the largest column number of the input matrix

DESCRIPTION

Blwarp is a two dimensional bilinear interpolation warping function. It first finds the four vertices of the squares that contain each of the points that we want to interpolate. The function *nnwarp* is used to find these vertices and the points are given by the matrices (rp+rf) and (cp+cf). Rp and cp define the mapping to be implemented.

The bilinear interpolation is performed in the following way:



$$P = (1 - cf)*(1 - rf)*P1 + (1 - rf)*cf*P2 + (1 - cf)*rf*P3 + cf*rf*P4$$

for all points P in the matrix.

AUTHOR

Maria C. Gutierrez

SEE ALSO

rotation(2),nnwarp(2)

NAME

compn – near neigbor comparison function

SYNOPSIS

{$library mx.pl }

compn (m:mtype; w:wtype):mtype; extern size;

TYPES

mtype = parallel array [lo1..hi1, lo2..hi2] of boolean;
wtype = array [0..size, 0..size] of 0..2;

DESCRIPTION

Compn compares the local neighborhood of each element of the boolean input matrix m with the window w. If a match occures then the result element is true, otherwise it is false. A zero in w matches with false in m, a one in w matches with true in m and a two in w is a don't care.

AUTHOR

A. P. Reeves

NAME

gaussian – gaussian pyramid: low pass filter

SYNOPSIS

{$library Pyramid.pl }

function gaussian(image:gtype; weight:real):rtype;
extern nrows ncols;

TYPES

gtype = parallel array [0..nrows, 0..ncols] of (real or integer);
rtype = parallel array [0..nrows, 0..ncols] of real;
itype = parallel array [0..nrows, 0..ncols] of integer;
btype = parallel array [0..nrows, 0..ncols] of boolean;

EXTERN CONSTANTS

nrows = The largest row number of the image matrix
ncols = The largest column number of the image matrix

VARS

id1,id2,up1,dn1,up2,dn2: itype
pyrmsk: btype;
Id1 and *id2* are two global index identifying matrices as created by twodid (see mat(2)). These must be initialized first. The other matrices specify transformations for managing pyramid data; see pyramid(2) for more information.

DESCRIPTION

Gaussian is a low pass filter function for pyramid structure images stored in a two dimensional matrix. The successive levels of the pyramid structure are stored in successive rows of the image matrix with the highest level image (1 pixel image) being located at position [0,0] in the input matrix. Each level is a low pass filtered copy of its predecessor and it is obtained by convolving the predecessor with a Gaussian weighting kernel. This kernel is generated from the weight parameter; if a gaussian kernel is desired the weight should be equal to 0.4.

The pyramid operations require several constant matrices; these are declared globally for efficiency. The global variables are generated with the functions *twodid* (see mat(2)), *pyrmskg* and *pyrgen* (see pyramid(2)).

AUTHOR

Maria C. Gutierrez

SEE ALSO

pyramid(2), xshift(2), xconv(2), mat(2), gather(2)

NAME

nnwarp – Near neighbor warping

SYNOPSIS

{$library nnwarp.pl }

function nnwarp(mx:pa; r:pi; c:pi; mask:pb):pa;extern lo1, hi1, lo2, hi2;

TYPES

pa = array [lo1..hi1,lo2..hi2] of btype;
pi = array [lo1..hi1,lo2..hi2] of itype;
pb = array [lo1..hi1,lo2..hi2] of boolean;
Where btype is any type and itype is an integer or subrange base type.

EXTERN CONSTANTS

lo1 = the smallest row number of the input matrix
lo2 = the largest row number of the input matrix
hi1 = the smallest column number of the input matrix
hi2 = the largest column number of the input matrix

VARS

id1,id2: pi;
Id1 and id2 are two global index identifying matrices as created by twodid (see mat(2)). These must be initialized before using nnwarp.

DESCRIPTION

Nnwarp is a two dimensional near neighbor warping function. The transformation implemented by nnwarp is as follows:

nnwarp[i,j] := mx[r[i,j],c[i,j]];

The r and c matrices contain the row and column indices, respectively, of where each element in nnwarp is to be obtained from in mx. i.e. r and c define the mapping to be implemented.

AUTHOR

Maria C. Gutierrez

SEE ALSO

mat(2)

NAME

laplacian – laplacian pyramid: band pass filter

SYNOPSIS

{$library Pyramid.pl }

function laplacian(image:gtype; weight:real):rtype;
extern nrows ncols;

TYPES

gtype = parallel array [0..nrows, 0..ncols] of (real or integer);
rtype = parallel array [0..nrows, 0..ncols] of real;
itype = parallel array [0..nrows, 0..ncols] of integer;
btype = parallel array [0..nrows, 0..ncols] of boolean;

EXTERN CONSTANTS

nrows = The largest row number of the image matrix
ncols = The largest column number of the image matrix

VARS

id1,id2,up1,dn1,up2,dn2: itype
pyrmsk: btype;
Id1 and *id2* are two global index identifying matrices as created by twodid (see mat(2)). These must be initialized first. The other matrices specify transformations for managing pyramid data; see pyramid(2) for more information.

DESCRIPTION

Laplacian is a band pass filter function for pyramid structure images stored in a two dimensional matrix. The successive levels of the pyramid structure are stored in successive rows of the image matrix with the highest level image (1 pixel image) being located at position [0,0] in the input matrix.

In the Laplacian pyramid, each level is a band pass filtered copy of its predecessor and it is obtained by the difference of two levels of the Gaussian pyramid. See gaussian(2) for more information about the Gaussian pyramid.

The pyramid operations require several constant matrices; these are declared globally for efficiency. The global variables are generated with the functions *twodid* (see mat(2)), *pyrmskg* and *pyrgen* (see pyramid(2)).

AUTHOR

Maria C. Gutierrez

SEE ALSO

Pyramid(2), pyramid(2), xshift(2), xconv(2), mat(2), gather(2)

NAME

lmaximum – local maximum filter function

SYNOPSIS

{$library lmean.pl }

function lmaximum(m:mtype; w:integer):mtype; extern shifttype;

TYPES

mtype = parallel array[il..ih,jl..jh] of btype;
shifttype = shift, lshift, lshiftg, crshift, or crshiftg
Where btype is integer or real

DESCRIPTION

Lmaximum computes the local maximum of the elements in the square (w*w) window. If the window goes beyond the input data then all values outside the data border are assumed zero. The maximum value is returned at the location of the central element of the window. For the case that the window size is even, the central element is located above and to the left of the center of the window.

The extern parameter shift can be selected according to the type of shift operation desired. If, for example, the local mean function is to be used on large arrays (exceeding the 128*128 array size of the MPP) then the lshift or lshiftg options can be used. In addition, there is a crshift and a crshiftg operation available. For more information on these functions consult the User's Manual.

AUTHOR

Marc Willebeek-LeMair

SEE ALSO

lmean(2), lmedian(2)

NAME

lmean – local mean filter function

SYNOPSIS

{$library lmean.pl }

function lmean(m: mtype): mtype; extern w, il, ih, jl, jh, shifttype;

TYPES

mtype = array[il..ih,jl..jh] of btype;
shifttype = shift, lshift, lshiftg, crshift, crshiftg
Where btype is integer or real

DESCRIPTION

Lmean computes the local mean of the elements in the square (w*w) window. The mean value is returned at the location of the central element of the window. For the case that the window size is even, the central element is located above and to the left of the center of the window. Along the border of the input array where the window contains elements outside the array boundary (of unknown value) the mean is set to zero. Therefore, the output array contains a border of zeroes half a window-length wide.

The extern parameter shift can be selected according to the type of shift operation desired. If, for example, the local mean function is to be used on large arrays (exceeding the 128*128 array size of the MPP) then the lshift or lshiftg options can be used. In addition, there is a crshift and a crshiftg operation available. For more information on these functions consult the User's Manual.

The window size (w) is specified as an extern parameter since it is used in the variable declaration of lmean as a dimension parameter.

AUTHOR

M. Willebeek-LeMair

SEE ALSO

lmaximum(2), lmedian(2)

NAME

lmedian – local median filter function

SYNOPSIS

{$library lmean.pl }

function lmedian(m: mtype; w: integer): mtype; extern il, ih, jl, jh, shifttype;

TYPES

mtype = array[il..ih,jl..jh] of btype;
shifttype = shift, lshift, lshiftg, crshift, or crshiftg
Where btype is integer or real

DESCRIPTION

Lmedian computes the local median of the elements in the square window of area wa. If the window goes beyond the input data then all values outside the data border are assumed zero. The median value is returned at the location of the central element of the window. For the case that the window size is even, the central element is located above and to the left of the center of the window.

AUTHOR

M. Willebeek-LeMair

SEE ALSO

lmaximum(1), lmean(1)

NAME

sort,rowsort,colsort – General two dimensional sort procedure

SYNOPSIS

{$library sort.pl }

procedure sort(var x:pd; var y:pd;);extern pb, n;
procedure rowsort(var x:pd; var y:pd;);extern pb, n;
procedure colsort(var x:pd; var y:pd;);extern pb, n;

TYPES

pd = array [lo1..hi1,lo2..hi2] of btype;
pb = array [lo1..hi1,lo2..hi2] of boolean;
Where btype is the base type of the data array to be sorted and n is the dimension of one side of the matrix. i.e., (hi1 - lo1 - 1). (n must be a power of 2)

VARS

id1,id2: pi;
Id1 and id2 are two global index identifying matrices as created by twodid (see mat(2)). These must be initialized before using sort.

DESCRIPTION

Sort is a general purpose two dimensional sorting procedure based on the bitonic sorting algorithm. It is designed for a parallel computer architecture which involves a square mesh connected processor array. The matrix to be sorted is the parameter x and the result is returned in y. X will be modified by this procedure, if this is not desirable for the given application then simply do not specify x to be a var parameter.

Rowsort is similar to sort except that each row of the matrix is individually sorted. *Colsort* sorts the columns of the matrix.

AUTHOR

A. P. Reeves

SEE ALSO

mat(2)

NAME

vshift – matrix spiral shift function

SYNOPSIS

{$library vshift.pl }

function vshift(a:atype; s:integer):atype; extern n, mtype;

TYPES

atype = array[il..ih,jl..jh] of btype;
mtype = array[il..ih,jl..jh] of boolean;
Where btype is any base type
n = the number of columns of m (jh-jl)

DESCRIPTION

Vshift performs a spiral shift on a matrix by a specified number of elements (s). That is, the last element in the first row of the matrix spirals around to the first element of the second row, and so forth. The last element of the last row is discarded. Zeroes are inserted at the first element of the first row. The following is an example of a 5*5 array (m), spiral shifted by 3 elements (s) using the vshift function.

1	2	3	4	5		0	0	0	1	2		4	5	6	7	8
6	7	8	9	10		3	4	5	6	7		9	10	11	12	13
11	12	13	14	15		8	9	10	11	12		14	15	16	17	18
16	17	18	19	20		13	14	15	16	17		19	20	21	22	23
21	22	23	24	25		18	19	20	21	22		24	25	0	0	0
		(a)						(b)						(c)		

(a) Original matrix m. (b) Vshift(m;s) using s = 3. (c) Vshift(m;s) using s = -3.

This function is useful in instances where a two dimensional array is being treated as a large vector.

AUTHOR

M. Willebeek-LeMair

SEE ALSO

vrotate(2)

NAME

vrotate – vector spiral rotate function

SYNOPSIS

{$library vshift.pl }

function vrotate(a:atype; s:integer):atype; extern n, mtype;

TYPES

atype = array[il..ih,jl..jh] of btype;
mtype = array[il..ih,jl..jh] of boolean;
Where btype is any base type
n = the number of columns of m (jh-jl)

DESCRIPTION

Vrotate performs a spiral rotate on a matrix by a specified number of elements (s). That is, the last element in the first row of the matrix spirals around to the first element of the second row, and so forth. The last element of the last row is inserted at the location of the first element of the first row. The following is an example of a 5*5 array (m), spiral rotated by 3 elements (s) using the vrotate function.

1	2	3	4	5		23	24	25	1	2		3	4	5	6	7
6	7	8	9	10		4	5	6	7	8		9	10	11	12	13
11	12	13	14	15		8	9	10	11	12		14	15	16	17	18
16	17	18	19	20		13	14	15	16	17		19	20	21	22	23
21	22	23	24	25		18	19	20	21	22		24	25	1	2	3
		(a)						(b)						(c)		

(a) Original matrix m. (b) Vrotate(m;s) using s = 3. (c) Vrotate(m;s) using s = -3.

This function is useful in instances where a two dimensional array is being treated as a large vector.

AUTHOR

M. Willebeek-LeMair

SEE ALSO

vshift(2)

PARALLEL PASCAL DEVELOPMENT SYSTEM: TUTORIAL

Anthony P. Reeves
School of Electrical Engineering
Cornell University
Ithaca, New York 14853

Abstract

This document demonstrates various features of the Parallel Pascal Development system which consists of a Parallel Pascal Translator and a library of support functions. A basic knowledge of the Parallel Pascal programming language is assumed. An example compilation is presented and the contents of the generated files is discussed. Advanced debugging procedures are considered.

Compiling a Program

The first step is to create a source Parallel Pascal file having a .pp extension with any text editor. An example program which is stored in a file called *square.pp* is shown below:

```
program square(input, output);
{$library mat }
const
     dim1 = 2;
     dim2 = 2;
type
     ar = array [1..dim1, 1..dim2] of integer;
var
     a:ar;
procedure writemx( mx:ar; fmt:integer); extern 1, dim2;
begin
     read(a);
     a := a * a;
     writemx(a, 4);
end.
```

To compile this program for a conventional computer simply type

pp square.pp

If the computer system has a Pascal interpreter in addition to a compiler for fast program development this can be selected with a *-i* flag; i.e.,

pp square.pp -i

Assuming that the first command is typed, selecting the local Pascal compiler, then the following messages will be output to the terminal

```
*** Pascal Library Processor & ***
*** Parallel Pascal Translator ***
Syntax Analysis Complete,
```

```
No Errors detected.
***      Pascal Compiler      ***
```

This indicates that the compilation was successful. The following files are created:

pplist	a listing of the Parallel Pascal Program
square.p	a Pascal version of the Parallel Pascal Program
square	a binary file ready for execution

When using the VMS operating system the file *square.p* may be named *square.pas*.

The listing file *pplist* has the following contents:

```
   1  program square(input, output);
   2  { library mat }
   3  const
   4     dim1 = 2;
   5     dim2 = 2;
   6  type
   7     ar = array [1..dim1, 1..dim2] of integer;
   8  var
   9     a:ar;
  10  procedure writemx( mx:ar; fmt:integer);
  11  (*$l-
*)(*$i+*)begin
  12     read(a);
  13     a := a * a;
  14     writemx(a, 4);
  15  end.
```

Syntax Analysis Complete, No errors detected.

In this program a single library *mat* is referenced which contains the procedure *writemx*. In the listing file the extern statement in line 10 is replaced by the library preprocessor with the body of the procedure writemx at line 11. The presence of an inserted body is indicated by the sequence (*$l-. . .*)(*$i+*). Since this is usually of no interest to the programmer the body itself is not listed. In this way all line numbers in the listing file correspond exactly with the line numbers of the source program.

Running a Program

The exact procedure for running a program varies from system to system. An example dialogue with a UNIX operating system is shown below for the program square. Input from the user is shown in italics.

```
% square
1 2 3 4
  1  4
  9 16
% square
1
9 8
5
```

```
 1 81
 72 25
% square < infile > ofile
%
```

These examples demonstrate the free format used for data input. The last command is an example of how, with UNIX, data may read from a file *infile* rather than the terminal and the result written to a file *ofile*.

Program Debugging

The usual cycle of events when errors are detected is to examine the listing file and then re-edit the source program. This procedure is slightly more difficult if an error is found in the body of a library function since these bodies are not listed. In general, errors in library functions are rare and are usually caused by an incorrect procedure declaration.

Consider the program square in which such an error has been made; i.e.,

```
program square(input, output);
{$library mat }
const
     dim1 = 2;
     dim2 = 2;
type
     ar = array [1..dim1, 1..dim2] of integer;
var
     a:ar;
procedure writemx( mx:ar; fmt:real); extern 1, dim2;
begin
     read(a);
     a := a * a;
     writemx(a, 4);
end.
```

On typing the command

pp square.pp

the result is:

```
*** Pascal Library Processor & ***
*** Parallel Pascal Translator ***
Syntax Analysis complete,
 1 errors detected.
***       No Compilation       ***
```

Only the file *pplist* is created; its contents are:

```
   1  program square(input, output);
   2  { library mat }
   3  const
```

```
   4      dim1 = 2;
   5      dim2 = 2;
   6   type
   7      ar = array [1..dim1, 1..dim2] of integer;
   8   var
   9      a:ar;
  10   procedure writemx( mx:ar; fmt:real);
  11   (*$l-
  11
  ****                        ^116
*)(*$i+*)begin
  12      read(a);
  13      a := a * a;
  14      writemx(a, 4);
  15   end.

Syntax Analysis Complete,   1 errors detected.
```

116: error in type of standard procedure parameter

The error has occurred on line 11 of the listing, i.e. within the body of the library procedure, but it is not clear what the cause is. In order to make library function bodies appear in the listing, specify the *-s* flag for the compiler; i.e.,

```
pp square.pp -s
```

The messages to the terminal will be the same as before; however, *pplist* will now contain the library procedure bodies as shown below:

```
   1  program square(input, output);
   2  { library mat }
   3  const
   4     dim1 = 2;
   5     dim2 = 2;
   6  type
   7     ar = array [1..dim1, 1..dim2] of integer;
   8  var
   9     a:ar;
  10  procedure writemx( mx:ar; fmt:real);
  11  {procedure writemx(mx;fmt:integer); extern lo1, hi1;}
  12  var
  13     i:integer;
  14  begin
  15     for i := 1 to dim2 do
  16       begin
  17         write(mx[i,]:fmt);
 ****                        ^116
  18         writeln;
  19       end
```

```
20  end;
21  begin
22     read(a);
23     a := a * a;
24     writemx(a, 4);
25  end.
```

Syntax Analysis Complete, 1 errors detected.

116: error in type of standard procedure parameter

The reason for the error is now clear. The format parameter *fmt* must be declared to be of type integer rather than real as shown in the comment on line 11. When using the *-s* option, line numbers in the listing file which follow the first library function no longer correspond to the line numbers in the source file.

Using the Library Preprocessor

Standard Pascal has no library facility; all subprograms i.e., procedures and functions, must be present in the source program. A library preprocessor was developed to allow the use of libraries without violating the rules of standard Pascal. The header line of a library subprogram is specified in the source program with an **extern** directive. The library preprocessor replaces the extern directive with the appropriate subprogram body. The type information for the library subprogram is extracted from the declaration statement in the source program. Therefore, library subprograms can be written to work with any user specified array type.

If a library subprogram is to be used for more than one array type in the same block, then a subprogram declaration statement for each unique argument type is necessary. Each unique version of the subprogram is identified by a user specified extension to the subprogram name in both declaration and usage.

For example, consider the ceiling function as defined below:

```
function ceiling(x:xtype) : rtype;
begin
   where x < 0.0 do
      ceiling := trunc(x)
   otherwise
      where x-trunc(x) = 0.0 do
         ceiling := trunc(x)
      otherwise
         ceiling := trunc(x)+1;
end;
```

The following program fragment illustrates how more than one version of this function could be specified for the library preprocessor.

```
. . .
type
      ar = array [1..10] of real;
      ai = array [1..10] of integer;
      br = array [1..8, 1..8] of real;
      bi = array [1..8, 1..8] of integer;
{$library math }
```

```
function ceiling.a(x:ar) :ai; extern;
function ceiling.b(x:br) :bi; extern;
var
    ax:ar; ay:ai; bx:br; by:bi;
begin
 . . .
    ay := ceiling.a(ax);
    by := ceiling.b(bx);
 . . .
```

A library file consists of the bodies of a set of library subprograms separated by their names preceded by a # symbol. The following is the contents of a library file, such as math.pl, which contains the two library subprograms used in this tutorial. The usage of system library subprograms and their location is given in section 2 of the Parallel Pascal Development System Manual. In addition, by convention, a descriptive comment line for the format of the subprogram header is included with the body of each library subprogram. For further information on the structure of the library files see extern(1) and for the usage of these subprograms see mat(2) and math(2).

```
#ceiling
{
function ceiling(x:real(array)) : integer(array); extern;
}
begin
   where x < 0.0 do
      ceiling := trunc(x)
   otherwise
      where x-trunc(x) = 0.0 do
         ceiling := trunc(x)
      otherwise
         ceiling := trunc(x)+1;
end;
#writemx
{
procedure writemx(mx:matrix;fmt:integer); extern lo1, hi1;
}
var
   i:integer;
begin
   for i := $3 to $4 do
     begin
       write(mx[i,]:fmt);
       writeln;
     end
end;
#end
```

For the Intrepid Explorer

If errors are reported after the translator these are generated by the local Pascal Compiler or linker. In this case the translator usually reports *No errors detected* then error messages appear on the terminal. The cause of these errors may either be an incorrect translation by the translator or by a limitation in the local Pascal compiler.

The translator generates a standard Pascal program and the intrepid explorer may wish to examine this to find the cause for more obscure errors. But beware, this program is typically six times longer than the original source program with many obscure variable names.

For example, the file *square.p* contains the following:

```
program square(input, output);

const
       dim1 = 2;
       dim2 = 2;
type
       ar =array[1..2,1..2] of integer;
     plltyp0 = ar;
     plltyp1 = array [1..2] of integer;
var
       a:ar;
     pllibool : boolean;
     pllvar0: ar;
     pllidx1: integer;
     pllidx0: integer;
     procedure writemx   ( mx:ar; fmt:integer);

var
   i:integer;
       pllvar0: plltyp1;
       pllidx0: integer;
begin
       for i:=1 to dim2 do
         begin
           for pllidx0:=1 to 2 do
             write(mx[i,pllidx0]:fmt);
           writeln;
         end
     end;
  begin
    for pllidx0:=1 to 2 do
      for pllidx1:=1 to 2 do
        read(a[pllidx0,pllidx1]);

    for pllidx0:=1 to 2 do
      for pllidx1:=1 to 2 do
        a[pllidx0,pllidx1]:=a[pllidx0,pllidx1]*a[pllidx0,pllidx1];
    writemx(a,4);
  end.
```

If still a glutton for more punishment, there is a debug compiler option {$c+ } which inserts the source program text as comment statements into the translated pascal file. This makes the pascal file easier to follow. For example, for full commented output add the debug option as shown below:

```
{$c+ }    { comment option }
program square(input, output);
{$library mat }
const
     dim1 = 2;
     dim2 = 2;
type
     ar = array [1..dim1, 1..dim2] of integer;
var
     a:ar;
procedure writemx( mx:ar; fmt:integer); extern 1, dim2;
begin
     read(a);
     a := a * a;
     writemx(a, 4);
end.
```

The file *square.p* generated by compilation will now contain additional comment statements as follows:

```
program square(input, output);

const
       dim1 = 2;
       dim2 = 2;
type
       ar =array[1..2,1..2] of integer;
     plltyp0 = ar;
     plltyp1 = array [1..2] of integer;
var
       a:ar;
     pllibool : boolean;
     pllvar0: ar;
     pllidx1: integer;
     pllidx0: integer;
     procedure writemx   ( mx:ar; fmt:integer);

var
   i:integer;
       pllvar0: plltyp1;
       pllidx0: integer;
begin
       for i:=1 to dim2 do
         begin
                              (*****>  TRANSLATE A PROCEDURE  <*****
                                    <<< ORIGINAL >>>
                                      write(mx[i,]:fmt);
                                    <<< TRANSLATED >>>   *)
```

```
        for pllidx0:=1 to 2 do
          write(mx[i,pllidx0]:fmt);
                        (***************************************)

        writeln;
      end
  end;
begin
                          (*****>  TRANSLATE A PROCEDURE  <*****
                                <<< ORIGINAL >>>
                             read(a);
                                <<< TRANSLATED >>>   *)
  for pllidx0:=1 to 2 do
    for pllidx1:=1 to 2 do
      read(a[pllidx0,pllidx1]);
                        (***************************************)

                         (*****>  TRANSLATE AN ASSIGNMENT  <*****
                                <<< ORIGINAL >>>
                           a:=a*a;
                                <<< TRANSLATED >>>   *)

  for pllidx0:=1 to 2 do
    for pllidx1:=1 to 2 do
      a[pllidx0,pllidx1]:=a[pllidx0,pllidx1]*a[pllidx0,pllidx1];
                        (***************************************)

                          (*****>  TRANSLATE A PROCEDURE  <*****
                                <<< ORIGINAL >>>
                             writemx(a,4);
                                <<< TRANSLATED >>>   *)
  writemx(a,4);
                        (***************************************)

end.
```

DATA MANIPULATIONS ON THE MASSIVELY PARALLEL PROCESSOR

Anthony P. Reeves and Cristina H. Francfort de Sellos Moura
School of Electrical Engineering
Cornell University
Ithaca, New York 14853

Abstract

The design of an effective processor interconnection network is a major problem in the design of highly parallel computer systems. A dense interconnection scheme such as the crossbar is prohibitively expensive for such systems; an affordable network must be designed which is capable of implementing the systems tasks without a significant loss in processor utilization. In this paper the mesh network of the Massively Parallel Processor (MPP) is analyzed for a number of important data manipulations. The MPP is a SIMD computer with 16384 processing elements connected in a 128 x 128 mesh. The MPP is an interesting system to study since it represents a minimal architecture in both processing element complexity and interconnection topology. The *transfer ratio*, which is the number of elemental processor operations required to execute a data manipulation, is introduced to measure the performance of the MPP for different data manipulations and to estimate the effectiveness of hardware enhancements. This type of analysis can be used to compare the performance of different interconnection schemes and may be applied to other highly parallel systems.

1. INTRODUCTION

In any highly parallel processor design a major consideration is the processor interconnection scheme. Processors must communicate with a speed which does not impede data processing; however, a general processor interconnection network is usually prohibitively expensive when a large number of processors are involved. A major design task is to design a restricted network which is adequate for the anticipated tasks for the system.

The Massively Parallel Processor consists of 16384 bit-serial Processing Elements (PE's) connected in 128 x 128 mesh [1]. That is each PE is connected to its 4 adjacent neighbors in a planar matrix. The two dimensional grid is one of the simplest interconnection topologies to implement, since the PE's themselves are set out in a planar grid fashion and all interconnections are between adjacent components. Furthermore, this topology is ideal for two dimensional filtering operations which are common to low level image processing such as small window convolution.

The PE's are bit-serial, i.e. the data paths are all one bit wide. This organization offers the maximum flexibility, at the expense of the highest degree of parallelism, with the minimum number of control lines. For example, as an alternative to the MPP consider 2048 8-bit wide PE's (on the MPP one chip contains 8 1-bit PE's). The 8-bit version would have a less rich set of instructions restricted to predefined byte operations while the bit-serial processors can process any data format. The advantage gained with the 8-bit system is that full processor utilization is achieved with arrays of 2048 elements while arrays of 16384 elements are required for full utilization of the MPP. The MPP PE is well matched to low level image processing tasks which often involve very large data arrays of short integers which may be from 1 to 16 bits.

In this paper the effectiveness of the MPP architecture for various interprocessor data manipulations is considered. The MPP offers a simple basic model for analysis since it involves just mesh interconnections and bit-serial PE's. Extensions to this scheme to speed up some manipulations are considered. The minimal architecture of the MPP is of particular interest to study, since any architecture modifications to improve performance would result in a more complex PE or a more dense interconnection strategy. The MPP is programmed in a high level language called Parallel Pascal [2]. The algorithms in this paper will be described in Parallel Pascal notation.

1.1 THE MPP PROCESSING ELEMENT

The MPP processing element is shown in Fig. 1. All data paths are one bit wide and there are 8 PE's on a single CMOS chip with the local memory on external memory chips. Except for the shift register, the design is essentially a minimal architecture of this type. The single bit full adder is used for arithmetic operations and the Boolean processor, which implements all 16 possible two input logical functions, is used for all other operations. The NN

Proceedings of the Nineteenth Hawaii International Conference on System Sciences (HICSS-19) 1986

select unit is the interface to the interprocessor network and is used to select a value from one of the four adjacent PE's in the mesh.

The S register is used for I/O. A bit plane is slid into the S registers independent of the PE processing operation and it is then loaded into the local memory by cycle stealing one cycle. The G register is used in masked operations. When masking is enabled only PE's in which the G register is set perform any operations; the remainder are idle. The masked operation is a very similar control feature in SIMD designs. Not shown in Fig. 1. is an OR bus output from the PE. All these outputs are connected (ORed) together so that the control unit can determine if any bits are set in a bitplane in a single instruction. On the MPP the local memory has 1024 words (bits) and is implemented with bipolar chips which have a 35 ns access time.

The main novel feature of the MPP PE architecture is the reconfigurable shift register. It may be configured under program control to have a length from 2 to 30 bits. Improved performance is achieved by keeping operands circulating in the shift register which greatly reduces the number of local memory accesses and instructions. It speeds up integer multiplication by a factor of two and also has an important effect on floating-point performance.

1.2 PERFORMANCE EVALUATION

In order to analyze the effectiveness of the interconnection network for different manipulations it is necessary to characterize the processing speed of the PE and the speed of the interconnection network. On the MPP both of these are data dependent; we have considered three representative cases: single-bit *Boolean* data, 8-bit *integer* data and 32-bit floating-point (*real*) data. For each of these data types we have estimated a *typical* time for an elemental operation. These estimates are of a reasonable order for this minimal PE architecture but are not very precise. For example, the instruction cycle time for a memory access and operation on the MPP is 100 ns. An elemental boolean operation may be considered to take 100 ns; however, it may be argued that an operation should involve two operands and have all variables in memory in which case three memory accesses (instructions) would require 300ns. For our analysis a two instruction (200 ns) model was used to represent Boolean instruction times. For the real and integer data a convenient number midway between the times for addition and multiplication was used. It should be remembered that elemental operations also include many other functions such as transcendental functions since these can be computed in times comparable to a multiplication on a bit-serial architecture. By adding a large amount of additional hardware to each PE it is possible to increase the speed of multiplication by 10 times or more [3].

For each of the data manipulations considered, times for the three different data types will be computed. The performance of the MPP for each manipulation will be indicated by the ratio of the data transfer time to an elemental PE operation on the same data type; this will be called the *transfer ratio*. One way to look at this ratio is the number of elemental data operations which must be performed between data transfers for the data transfers not to be the dominant cost for the algorithm. On the MPP data may be shifted between adjacent PE's in one instruction time (100 ns.) concurrently with a PE processing instruction.

For many applications the physical dimensions of the parallel hardware are smaller than the dimensions of the array to be processed. In this case the data array is processed as a set of blocks. An extension of the data manipulation algorithms to deal with this situation is discussed.

The program and algorithm examples given in this paper use the Parallel Pascal notation. This notation involves the following five extensions to standard Pascal:

1) expressions involving whole arrays are permitted; for mixed operations between a scalar and an array the scalar is replicated to form a conformable array.
2) the **where - do - otherwise** control statement is available. This statement is a parallel version of the **if - then - else** statement; the control expression must evaluate to a Boolean array. All array assignments within the controlled statements must be conformable with the control array and are masked by it.
3) the functions *any* and *min* are the array reduction functions or and minimum respectively.
4) array indices may be elided for subarray selection. For example, consider a matrix *a*; *a*[i,] specifies the i'th row, *a*[,j] specifies the j'th column and *a*[,] specifies the whole matrix.
5) basic parallel data manipulation operations, including machine primitive manipulations, are available as built in functions *shift*, *rotate*, *expand* and *transpose*.

2. SHIFT AND ROTATE OPERATIONS

The only permutation function which is directly implemented by the MPP is the near neighbor rotate (or shift). The direction of the rotation may be in any of the four cardinal directions. In Parallel Pascal the main permutation functions are multi-element rotate and shift functions; other permutations are built on these primitives.

The rotate function takes as arguments the array to be shifted and a displacement for each of the arrays dimensions. For example consider a one dimensional array *a* specified by

a: **array** [0..n] of integer;

The rotate statement

```
a := rotate( a, 5);
```

is equivalent to

```
for i := 0 to n do
    a[i] := a[(i + 5) mod (n + 1)];
```

The rotation utilizes the toroidal end around edge connections of the mesh. The *shift* function is similar except that the mesh is not toroidally connected and zeroes are shifted into elements at the edge of the array; therefore, the shift function is not a permutation function in the strict sense. The concept of the rotate and shift functions extends to n dimensions; on the MPP the last two dimensions of the array correspond to the parallel hardware dimensions and are executed in parallel, higher dimension operations are implemented in serial. The cost of the rotate function is dependent on the distance rotated. It also depends on the size of the data elements to be permuted.

The transfer ratios for the shift operation are given in Table 1. Ratios are given for shift distances of 1 and 64 elements; 64 is the largest shift which will normally be required in a single dimension on a 128 x 128 matrix since a shift of 65 can be obtained with a rotate of -63 and a mask operation. The worst case figures for a two dimensional shift is 64 in each direction; i.e., twice the figures given in Table 1.

For single element shifts the interconnection network is more than adequate for all data types. For maximum distance shifts the ratio of 33 for Boolean data could cause problems for some algorithms but the situation is much better for real data.

3. MATRIX MAPPING

Table 1: Cost for Shift and Rotate Operations

Shift distance	Operation cost in μs.			Transfer Ratio		
	Boolean	integer	real	Boolean	integer	real
1	0.2	1.6	6.4	1.0	0.32	0.16
64	6.5	51	210	33	10	5.2

In this section the arbitrary mapping of data from one matrix to another matrix with the same dimensions is considered. Since there are no restrictions placed on the mapping it includes the worst case mapping which can occur. The mapping of a matrix a is specified by two coordinate matrices c and r which have similar dimensions to a. The permuted matrix b also has the same dimensions as a. For a matrix element b[i,j] the corresponding elements r[i,j] and c[i,j] specify the row and column indices respectively of where the related element of a is located. That is, the mapping is specified by

```
b[i,j] := a[ r[i,j], c[i,j] ]
```

More formally, the data arrays involved in the permutation are specified by:

```
a,b : array [1..nrow,1..ncol] of data;
{where data is any base type}
r : array [1..nrow,1..ncol] of 1..nrow;
c : array [1..nrow,1..ncol] of 1..ncol;
```

For the MPP the last two dimensions of the array (nrow and ncol) must both be 128.

For the following algorithms, and in general for the MPP, two PE identifying matrices *idr* and *idc* are precomputed. They contain the following:

```
for i := 1 to nrow do  idr[i,] := i;
for j := 1 to ncol do  idc[,j] := j;
```

These are used to select subsets of PE's for an operation. For example a mask matrix which is true only at element i,j can be specified by the expression ((idr = i) and (idc = j)).

3.1 A SIMPLE MAPPING ALGORITHM

A simple, but slow, algorithm to achieve an arbitrary permutation is to slide a over all the possible positions of b, assigning the specified elements of a to each element of b when they are in the correct position.

The first step is to compute matrices *rr* and *rc* which specify the distance data must be moved rather than the absolute location of the data.

```
rr := (r-idr) mod nrow;
rc := (c-idc) mod ncol;
```

Data is then loaded into the result matrix when it has been moved by the correct amount.

```
for i := 1 to nrow do
    begin
        for j := 1 to ncol do
            begin
                where (rr = i) and (rc = j) do
                        b := a;
                a := rotate(a, 0, 1);
            end;
        a := rotate(a, 1, 0);
    end;
```

In this algorithm, data is only explicitly moved in the upward and left directions; data which must be moved in the right or down-

ward directions is taken care of by the toroidal end connections specified by the *rotate* primitive operation.

This algorithm involves $O(n^2)$ operations for an n x n matrix. Each iteration requires each PE to compute two 7-bit comparisons (which are shown in the where statement); these take 1.4 μs. The total cost for 16384 iterations is shown in Table 2. It is of no surprise that the costs are so high for arbitrary mappings on the very restricted interprocessor network of the MPP. However, table 2 is useful in that it provides an upper bound for arbitrary data mappings.

A final problem with this algorithm is that it has a complexity of slightly worse than $O(n^4)$. For example, a 256 x 256 matrix requires 16 times the computation required for a 128 x 128 matrix.

4. DATA BROADCAST

Table 2 : Cost for the Simple Mapping Algorithm

Total Cost	Operation cost in μs.			Transfer Ratio		
	Boolean	integer	real	Boolean	integer	real
Original	29000	75000	230000	150000	15000	5800
Optimized	18000	64000	220000	90000	13000	5500
With shift register	16000	50000	170000	82000	10000	4200

A frequently used data manipulation is to broadcast one element of data to a large number of PE's. In this section two broadcast operations are considered. The first is global data broadcast in which a single element is broadcast to all elements of a matrix or array. In Parallel Pascal this is represented by:

```
b := a[i,j];
```

The second type is a row or column broadcast in which a single row (or column) is broadcast to all rows (or columns) in the result matrix. More formally, these transformations for the k'th row or column may be represented in Parallel Pascal by

```
for i := 1 to nrow do   { row distribution }
  b[i,] := a[k,];

for j := 1 to ncol do   { column distribution }
  r[,j] := a[,k];
```

In fact the function *expand* is included in the Parallel Pascal language so that this important transformation may be specified without a loop; e.g.

```
b := expand(a[k,],1);     { row distribution }
```

For global broadcast the first step is to transfer the value to be distributed from the matrix to the control unit. The MPP has a global *or* hardware mechanism which can detect if any bit in a bitplane is set. This mechanism can also be used to extract the maximum or minimum value in a matrix in a similar amount of time. To extract a value from a matrix first set a mask which is true only at the PE of interest (14 instructions). Then *and* this mask with each bit plane of the data matrix and detect if any value is set with the global *or* mechanism (n instructions where n is the number of bits in the data). This value is then distributed to all PE's with the instruction stream (which may be considered as a global broadcast mechanism); this requires n instructions.

A simple algorithm to do row (or column) distribution is to take the row (or column) and distribute it across the matrix in 128 steps. For example, the following algorithm will distribute the k'th row of a Boolean matrix a.

```
mask: array [1..nrow, 1..ncol] of boolean;

mask := idr = k;
b := a and mask;
for i = 1 to nrow do
    b := b or rotate(b, 1, 0);
```

For integer and real data the above algorithm can be repeated for each data bit plane. This algorithm has complexity $O(mn)$ where n is the number of bits in each data element and m is the length of the row. On the MPP a faster solution is possible involving the PE shift registers. All bits of the row to be distributed are first shifted in adjacent rows of a single bitplane then, as the data passes by a PE, the PE loads the data into its shift register. The complexity of this algorithm is $O(n+m)$. The costs of different distribution algorithms are given in Table 3.

Table 3. : Cost for the Distribution Algorithms

Algorithm	Operation cost in μs.			Transfer Ratio		
	Boolean	integer	real	Boolean	integer	real
Global	0.4	3.2	13	2	0.64	0.32
Row	14	100	410	68	21	10
Row with shift register	14	16	21	68	3.2	0.52

0 1 2 3 4 5 6 7 input vector

0 4 1 5 2 6 3 7 shuffled vector

5. SHUFFLES AND PYRAMIDS

In this section shuffle permutations are considered. In addition to their usual applications to algorithms such as the Fast Fourier Transform (FFT) they are also the basic permutation used to manipulate pyramid data structures. The shuffle permutation in one dimension is illustrated below; the vector is split into two equal halves and then interleaved as in shuffling a pack of cards.

The two-dimensional shuffle of a matrix may be considered as a combination of two orthogonal one-dimensional shuffles; first all the rows are shuffled and then the columns. The algorithms we have considered can also be divided into two half shuffle operations; for example, a half shuffle may map only elements 0, 1, 2 and 3 of the above input vector.

Detailed algorithms will only be given for a half shuffle operation; a full shuffle requires two half shuffles and a two-dimensional shuffle requires four half shuffles. The inverse shuffle permutation is also of interest. The shuffle algorithms can also perform inverse shuffles for a similar cost.

Operations on pyramid data structures are of interest to some image processing applications. A pyramid may be considered to be a connected set of multiresolution matrices. At the apex is one element, at the next level there a four elements in a 2 x 2 matrix each connected to the apex. At the next level there is a 4 x 4 matrix which may be considered as a disjoint set of four 2 x 2 submatrices with each submatrix being connected to one of the elements in the matrix at the next higher level. In general, the k'th level is a $2^k \times 2^k$ matrix with each element being associated with one element in the level above and four in the level below. On the MPP, the top 6 levels of a pyramid may be stored in a single matrix (the pyramid has a 64 x 64 base). Masked shift operations are used to perform shifts in all or a selected subset of levels simultaneously. The other fundamental operation is to move data up or down the levels of the pyramid. The two-dimensional half inverse shuffle and two-dimensional shuffle are the respective permutations for these pyramid operations.

Two half shuffle algorithms are considered; the first is the drop algorithm.

```
d := -1;  { drop algorithm }
md := 2;
mask := false;
mask [0,] := true;
b [0,] := a [0,];

for i := 1 to nrow div 2 - 1 do
    begin
        a := shift (a,d, 0);
        mask := shift (mask, md, 0);
        where mask do
            b := a;
    end;
```

In this algorithm the input matrix a is slid over the PE's and when the correct elements are aligned with their destinations they are *dropped* into them. The specific half shuffle performed is determined by the setup parameters. For example, if d is set to 1 and md is set to -1 then an inverse half shuffle will be implemented.

The second algorithm is called the take algorithm.

```
mask := false;  { take algorithm }
mask [nrow div 2 - 1] := true;
for i := 1 to nrow div 2 - 1 do
    begin
        where mask do
            b := a;
        b := shift (b, -1 , 0);
        mask := shift (b, 1, 0);
    end;
b [0,] := a [0,];
```

For this algorithm the result matrix b is slid over the PE's. When it is at the correct location to receive data it *takes* from the PE before passing on to its final destination.

The cost of these algorithms is shown in table 4. These algorithms have a similar performance when the PE architecture is simple but the take algorithm is better when the shift register is used. This is because the matrix being moved and modified is the same for this algorithm (b) and it can be efficiently operated on when stored in the shift register.

6. MATRIX WARPING

In many image processing applications a *warping* or rubber sheet distortion of a matrix is needed. The warp exhibits a locality property which is characteristic of many important data manipulations. That is, the adjacency between neighboring matrix elements is loosely maintained by the transform. Therefore, many elements will be moved by similar amounts; i.e., the distance to be moved matrices (rr and rc as defined in section 3) will contain many of the same or similar valued elements.

We have developed a heuristic algorithm which attempts to only make moves which are needed by the particular data manipulation; this is in contrast to the simple mapping algorithm of section 3 which proceeds through all n^2 possible data displacements.

The algorithm first slides (rotates) a as many locations up and left as possible such that future backtracking will not be necessary. If any element of a is correctly positioned over b (i.e. $(rr = 0)$ and

Table 4. : Half Shuffle Cost

Algorithm	Operation cost in μs.			Transfer Ratio		
	Boolean	integer	real	Boolean	integer	real
Take or Drop	32	220	830	160	45	21
Drop with SR	32	170	630	160	35	16
Take with SR shift register	32	110	420	160	23	11

(*rc* = 0)) then *b* is updated. Otherwise, *atr*, which is a copy of the current version of *a*, is slid in the upwards direction until all outstanding elements of *b*, for which the current *rc* = 0, are satisfied. The algorithm then shifts as far as possible up and left again and repeats the above procedure until all elements of the result mask are false, i.e. *b* is complete.

The following variables are used in the algorithm :

Variable declaration

mask,masktr : array [1_*nrow*, 1_*ncol*] of boolean;
atr : array [1_*nrow*, 1_*ncol*] of data;
rrt : array [1_*nrow*, 1_*ncol*] of 0_*nrow*;
ri, rit, lastrit : 0_*nrow*;
ci : 0_*ncol*;

Variable functions:

mask : the result mask, true values indicate elements of *b* which have not yet received the correct element of *a*.
ri, ci : row and column distances for the up-left move.
masktr : a version of *mask* to process one column.
rit : a version of *ri* used to process one column.
atr : a version of *a* used to process one column.
rrt : a version of *rr* used to process one column.
lastrit : the last value of *rit*.

The Parallel Pascal version of the heuristic algorithm is as follows:

```
lastrit :=0;
b := a;
mask := (rr <> 0) or (rc <> 0);

while any(mask, 1 ,2) do
   begin  { iterate until the permutation is complete }
      ri := min(rr, 1, 2);
      ci := min(rc, 1, 2);
      a := rotate(a, ri, ci); { move up and left as far as possible }
      rr := rr - ri;
      rc := rc - ci;
      masktr := (rr = 0) and (rc = 0);
      if any(masktr, 1, 2) then  { satisfy elements for the
                                   current position }
         atr := a
      else
         begin     {satisfy each element for the given column}
            where rc = 0 do
               rrt := rr
            otherwise
               rrt := nrow;
            rit := min(rrt, 1, 2);
            masktr := rrt = rit;
            { the next seven statements implement }
            { the statement atr = rotate (a, rit, 0) }
            { but also take advantage of the previous shifts }
            if ci <> 0 then
               begin
                  atr := a;
                  lastrit := 0;
               end;
            atr := rotate( atr, rit - lastrit, 0 );
            lastrit := rit;
         end;
      where masktr do {update b for the current location of a}
         begin
            b := atr;
            rr := nrow;
            rc := ncol;
            mask := false;
         end;
end;
```

This algorithm is bounded by n^2 iterations. However, this must be considered a loose bound since we currently do not know a permutation which would require all n^2 iterations.

The cost of this algorithm per iteration is approximately 10 times the cost of the simple mapping algorithm. However, for mappings which have very good locality properties, or for which not many elements need to be mapped, the mapping may be achieved in a very small number of iterations. It is difficult to represent the cost of this algorithm by any single number since its performance is so dependent on the specified data mapping. For a more detailed analysis of this algorithm and its applications see [4].

7. LARGE ARRAYS

Frequently the data to be processed by a parallel processor will be in the format of arrays which exceed the fixed range of parallelism of the hardware. Therefore, it is necessary to have special algorithms that will deal with large arrays by breaking them down into blocks manageable by the hardware, without loosing track of the relationships between different blocks.

One scheme, which is frequently used on the MPP, is to partition the large array into blocks which are conveniently stored in a four dimensional array. The range of the first dimension of this array specifies the number of blocks in each row of the large matrix and the range of the second dimension specifies the number of blocks in each column. Given a conceptual large matrix

mx : array [0_x,0_y] of btype;

which is to be stored in an array *a* of type

```
array [1..n, 1..m, 1..p, 1..q] of btype;
```

Element i,j of the large matrix is mapped into the array a as specified by

$$mx[i,j] = a\,[1+i \text{ div } p,\ 1+j \text{ div } q,\ 1+i \bmod p,\ 1+j \bmod q]$$

For example, a 512 x 256 matrix could be stored in eight blocks as:

```
la : array [1..4,1..2,1..128,1..128] of real;
```

This data structure allows blocks to be manipulated independently. However, it still preserves the positional relationships of those blocks in the original large matrix.

To simplify the manipulation of large arrays on the MPP, two Parallel Pascal library functions *lrotate* and *lshift* have been developed. These functions take an array argument and two displacement arguments, like the primitive matrix rotate and shift functions, however, in this case the array argument is a four dimensional array which is treated like a conceptually large matrix.

Many programs can be converted to operate on blocked rather than conventional matrices by simply replacing all instances of rotate and shift with lrotate and lshift respectively. This is true for the data manipulation algorithms presented; however, except for near neighbor type algorithms, a different strategy is usually more efficient. The transfer ratios for for different sized large arrays are shown in Table 5. It should be noted that not much processing can be done with 512 x 512 real matrices on the current MPP since a single matrix requires half of the local memory.

Single element shifts have slightly worse values for large arrays due to the small overhead involved in moving data at the edges of blocks to other blocks. Broadcast operations have better transfer ratios for large arrays since once a block has been set up it can be stored in several result blocks. The shuffle algorithm for each block of a large matrix is essentially the same as for a small matrix (simple masked swapping of matrix halves does occur), the transfer ratio is not significantly changed for large arrays. On the other hand, for pyramid data structures the data storage beyond level seven becomes much simpler and more efficient. The transfer ratio in this case is greatly improved for large arrays.

The cost of the simple mapping algorithm has a complexity of $O(n^4)$ therefore the transfer ratios are four times higher for the 256 x 256 matrix and 16 times higher for a 512 x 512 matrix. In most practical cases the heuristic algorithm will be far superior to the simple algorithm; however, for a truly random mapping there is no known efficient solution.

For the heuristic mapping algorithm a good strategy for large arrays is to scan through the result blocks and perform operations on only the input blocks that contribute to the current result block being processed. This algorithm is shown below.

```
var
  la,lb: array [1..n,1..m,1..nrow,1..ncol] of data;
  lr,lc: array [1..n,1..m,1..nrow,1..ncol] of index;

begin
  for i = 1 to n do
    for j = 1 to m do
      begin {process each result block}
        rb := 1 + lr[i,j] div nrow;
        cb := 1 + lc[i,j] div ncol;
        ro := 1 + lr[i,j] mod nrow;
        co := 1 + lc[i,j] mod ncol;
        for k = 1 to n do
          for l = 1 to m do
            begin {consider each input block}
              maskb := (rb = k) and (cb = l);
              if any(maskb, 1, 2) then
                where maskb do
                  lb[i,j] := perm2 (la [k,l] ,
                      ro, co, maskb);
            end;
      end;
end;
```

Perm2 is the heuristic algorithm presented previously with the modification that the initial mask value is passed as an argument. That is, only elements selected by the mask are permuted. An additional speedup is achieved by this since the heuristic works much better when only a subset of elements are to be permuted.

8. CONCLUSION

The transfer ratio has been introduced as a performance measure for data manipulations on parallel processors. This technique may be applied to almost any parallel architecture. It is expected that the results obtained for the MPP would be very similar to those obtained for other like architectures such as the Distributed Array Processor (DAP) [5] or NCR's GAPP processor chip which contains 72 PE's with local memory [6]. The results given in Table 5 could be used by programmers to predict the performance of algorithms on the MPP.

For the MPP, the results indicate that, although arbitrary data mappings may be very costly, some important data manipulations can be done very efficiently. The shift register, which has a 2 times speedup factor for multi-bit arithmetic also has a significant effect on the implementation of several of the multi-bit data manipulations studied. Especially interesting is the improvement of over 10 times for real data distribution. The shuffle cannot be implemented fast enough for efficient FFT implementation; however, other data mapping strategies for the FFT are well known which

Table 5: Transfer Ratios for Different Data Manipulations and Array Sizes

Data Manipulation	Array Size								
	128 x 128			256 x 256			512 x 512		
	Boolean	integer	real	Boolean	integer	real	Boolean	integer	real
Data Shift									
a) 1 element	1.0	0.32	0.16	2.0	0.5	0.24	2.0	0.5	0.24
b) worst case	33	10.2	5.2	33	10	5.2	33	10	5.2
Broadcast									
a) Global	2	0.64	13	0.88	0.28	0.14	0.59	0.20	0.09
c) Row with shift register	68	3.2	0.52	35	1.68	0.30	18.2	0.92	0.19
Shuffle (2-dimensional)	640	90	42	640	90	42	640	90	42
Pyramid (up or down)	320	45	21	58	4.3	3.5	16	1.2	0.98

have a much more efficient implementation on the MPP.

On the DAP row and column distribution is implemented directly by special hardware buses. For the MPP we can see from Table 3 that no advantage would be gained from this hardware for real data operations and possibly very little advantage for integer operations. A second architecture modification that has been recently proposed by several researchers is to directly implement the pyramid layer structure in hardware. From Table 5 we can see that the MPP already implements pyramid operations on large pyramids very efficiently; therefore, there is very little advantage to be gained from the addition of special hardware.

REFERENCES

1. K. E. Batcher, "Design of a Massively Parallel Processor," *IEEE Transactions on Computers* C-29(9) pp. 836-840 (September 1981).
2. A. P. Reeves, "Parallel Pascal: An extended Pascal for Parallel computers," *Journal of Parallel and Distributed Computing* 1 pp. 64-80 (1984).
3. A. P. Reeves, "The Anatomy of VLSI Binary Array Processors," in *Languages and Architectures for Image Processing*, ed. M. J. B. Duff and S. Levialdi, Academic Press (1981).
4. A. P. Reeves and C. H. Moura, "Permutation and Rotation Functions for the Massively Parallel Processor," in *Computing Structures and Image Processing*, ed. K. Preston, Academic Press (in press).
5. R. W. Gostick, "Software and Algorithms for the Distributed-Array Processor," *ICL Technical Journal*, pp. 116-135 (May 1979).
6. NCR Corporation, *Geometric Arithmetic Parallel Processor*, NCR, Dayton, Ohio (1984).

THE MASSIVELY PARALLEL PROCESSOR:

A HIGHLY PARALLEL SCIENTIFIC COMPUTER

Anthony P. Reeves

School of Electrical Engineering
Cornell University
Ithaca, New York 14853

INTRODUCTION

The Massively Parallel Processor (MPP) [1, 2] is a highly parallel scientific computer which was originally intended for image processing and analysis applications but it is also suitable for a large range of other scientific applications. Currently the highest degree of parallelism is achieved with the SIMD type of parallel computer architecture. With this scheme a single program sequence unit broadcasts a sequence of instructions to a large number of slave Processing Elements (PE's). All PE's perform the same function at the same time but on different data elements; in this way a whole data structure such as a matrix can be manipulated with a single instruction. The alternative highly parallel organization, the MIMD type, is to have an instruction unit with every PE. This scheme is much more flexible but also much more complex and expensive.

Computers based on the SIMD scheme are usually very effective for applications which have a good match to the constraints of the architecture. Furthermore, they are usually also extensible in that it is possible to increase the performance for larger data structures by simply increasing the number of PE's.

In order to utilize the features of a SIMD system, as with all computer designs, it is important for the programmer to have some knowledge of the underlying architecture; for example, it is important to know that some matrix operations have the same cost as a scalar operation. For these systems a special programming environment is usually used and, in general, serial programs designed for conventional serial computers must be reformulated for highly parallel architectures. The MPP is programmed in a high level language called Parallel Pascal [3]. Therefore, the main advantage of SIMD systems is a much more cost effective method for doing scientific computing than conventional computer or supercomputer systems. The major disadvantage is that the user must become familiar with a new kind of programming environment.

A major consideration in the design of a highly parallel computer architecture is the processor interconnection network. Processors must communicate with a speed which does not impede data processing; however, a general processor interconnection network is usually prohibitively expensive when a large number of processors are involved. A major design task is to design a restricted network which is adequate for the anticipated tasks for the system. The mesh interconnection scheme, used on the MPP, is simple to

implement and is well suited to a large number of image processing algorithms.

The first section of this chapter describes the architecture of the MPP. Then the convenient data structers which can be manipulated by the MPP are outlined and the high level programming environment is discussed. The performance of the processor interconnection network for important data manipulations is considered in detail since this indicates which algorithms can be efficiently implemented on the MPP. Finally, some current applications of the MPP are outlined.

The Impact of Technology

The implementation of highly parallel processors is made possible with VLSI technology. The MPP was designed with the technology available in 1977. The custom processor chip has 8000 transistors on an area of 235 x 131 mils and contains 8 bit-serial PE's. It is mounted in a 52 pin flat pack and requires 200 mW at 10 MHz; 5 micrometer design rules were used. The SIMD mesh architecture can directly take advantage of the ongoing major advances in VLSI technology. A number of more advanced chips have been developed since the MPP design. For example, the GAPP chip [4] developed with the technology of 1982, has 72 bit-serial PE's, each having 128 bits of local memory. This chip requires 500 mW at 10 MHz. ITT [5] is predicting that by 1987 they will be able to make 16 16-bit PE's (with four spare) on a single chip with each PE having 1k words of local memory. This chip would use 1.25 micrometer design rules and would involve 600,000 transistors on an area of 450 x 600 mils. For the more distant future, advantage can be taken of wafer scale integration as soon as it becomes economically available. Techniques for dealing with the fault tolerance needed with such a technology have already been considered [6].

THE MPP ARCHITECTURE

The Massively Parallel Processor consists of 16384 bit-serial Processing Elements (PE's) connected in 128 x 128 mesh [1]. That is each PE is connected to its 4 adjacent neighbors in a planar matrix. The two dimensional grid is one of the simplest interconnection topologies to implement, since the PE's themselves are set out in a planar grid fashion and all interconnections are between adjacent components. Furthermore, this topology is ideal for two dimensional filtering operations which are common to low level image processing such as small window convolution.

The PE's are bit-serial, i.e. the data paths are all one bit wide. This organization offers the maximum flexibility, at the expense of the highest degree of parallelism, with the minimum number of control lines. For example, as an alternative to the MPP consider 2048 8-bit wide PE's (on the MPP one chip contains 8 1-bit PE's). The 8-bit version would have a less rich set of instructions restricted to predefined byte operations while the bit-serial processors can process any data format. The advantage gained with the 8-bit system is that full processor utilization is achieved with arrays of 2048 elements while arrays of 16384 elements are required for full utilization of the MPP. The MPP PE is well matched to low level image processing tasks which often involve very large data arrays of short integers which may be from 1 to 16 bits.

The effectiveness of the MPP architecture for various interprocessor data manipulations is considered. The MPP offers a simple basic model for analysis since it involves just mesh interconnections and bit-serial PE's. The minimal architecture of the MPP is of particular interest to study, since any architecture modifications to improve performance would result in a more complex PE or a more dense interconnection strategy.

The MPP Processing Element

The MPP processing element is shown in Fig. 1. All data paths are one bit wide and there are 8 PE's on a single CMOS chip with the local memory on external memory chips. Except for the shift register, the design is essentially a minimal architecture of this type. The single bit full adder is used for arithmetic operations and the Boolean processor, which implements all 16 possible two input logical functions, is used for all other

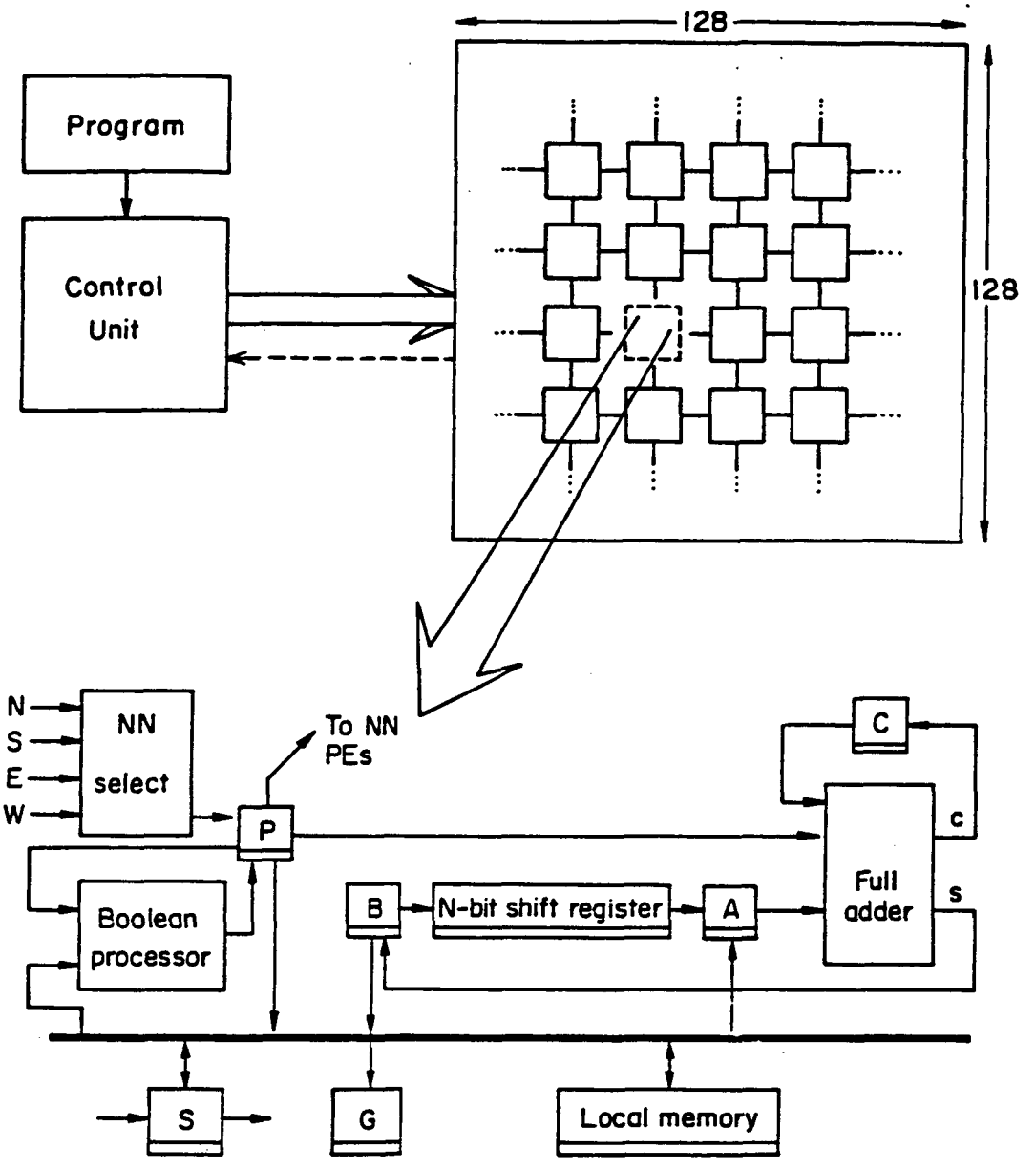


Figure 1. The MPP Processing Element

operations. The NN select unit is the interface to the interprocessor network and is used to select a value from one of the four adjacent PE's in the mesh.

The S register is used for I/O. A bitplane is slid into the S registers independent of the PE processing operation and it is then loaded into the local memory by cycle stealing one cycle. The G register is used in masked operations. When masking is enabled only PE's in which the G register is set perform any operations; the remainder are idle. The masked operation is a very common control feature in SIMD designs. Not shown in Fig. 1. is an OR bus output from the PE. All these outputs are connected (ORed) together so that the control unit can determine if any bits are set in a bitplane in a single instruction. On the MPP the local memory has 1024 words (bits) and is implemented with bipolar chips which have a 35 ns access time.

The main novel feature of the MPP PE architecture is the reconfigurable shift register. It may be configured under program control to have a length from 2 to 30 bits. Improved performance is achieved by keeping operands circulating in the shift register which greatly reduces the number of local memory accesses and instructions. It speeds up integer multiplication by a factor of two and also has an important effect on floating-point performance.

Array Edge Connections

The interprocessor connections at the edge of the processor array may either be connected to zero or to the opposite edge of the array. With the latter option rotation permutations can be easily implemented. This is particularly useful for processing arrays which are larger than the dimensions of the PE array. A third option is to connect the opposite horizontal edges displaced by one bit position. With this option the array is connected in a spiral by the horizontal connections and can be treated like a one-dimensional vector of 16384 elements.

The MPP Control Unit

A number of processors are used to control the MPP processor Array; their organization is shown in Fig. 2. The concept is to always provide the array with data and instructions on every clock cycle. The host computer is a VAX 11/780; this is the most convenient level for the user to interact since it provides a conventional environment with direct connection to terminals and other standard peripherals. The user usually controls the MPP by developing a complete subroutine which is down loaded from the VAX to the main control unit (MCU) where it is executed. The MCU is a high speed 16-bit minicomputer which has direct access to the microprogrammed array control unit (ACU). It communicates to the ACU by means of macro instructions of the form "add array A to array B". The ACU contains runtime microcode to implement such operations without missing any clock cycles. A first in-first out (FIFO) buffer is used to connect the MCU to the ACU so that the next macro operation generation in the MCU can be overlapped with the execution in the ACU. A separate I/O control unit (IOCU) is used to control input and output operations to the processor array. It controls the swapping of bitplanes between the processor array and the staging memory independent of the array processing activity. Processing is only halted for one cycle in order to load or store a bitplane.

I/O and the MPP Staging Memory

The staging memory is a large data store which is used as a data interface between peripheral devices and the processor array; it provides two main functions. First, it performs efficient data format conversion between the data element stream which is most commonly used for storing array data to the bitplane format used by the MPP. Second,

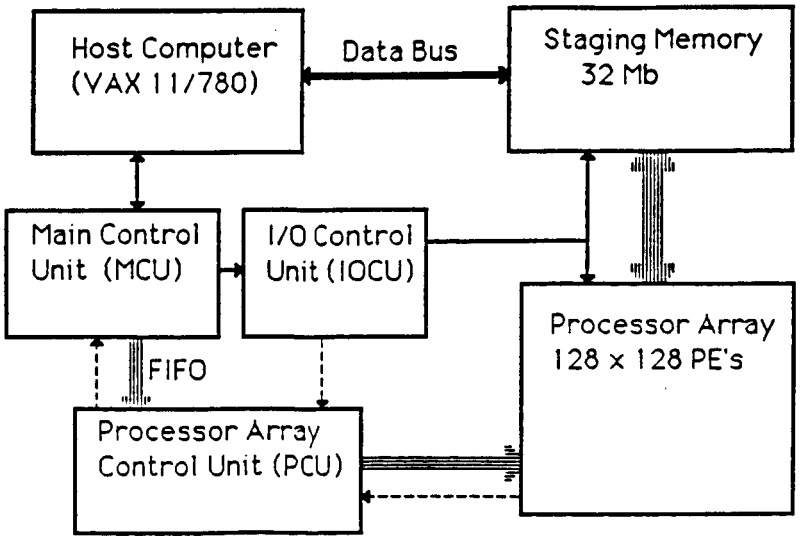


Figure 2. MPP System Organization

it provides space to store large (image) data structures which are too large for the processor array local memory.

The I/O bandwidth of the processor array is 128 bits every 100 ns; i.e., 160 Mbytes/second. When fully configured with 64Mb of memory the staging memory can sustain the MPP I/O rate. Currently the MPP is configured with 32 Mb and can sustain half the optimal I/O rate. In addition to data reformating, hardware in the staging memory permits the access of 128 x 128 blocks of data from arbitrary locations in a large (image) data structure. This feature is particularly useful for the spooling scheme which is outlined in the following section.

MPP DATA STRUCTURES

The 128 x 128 array is the equivalent to the natural *word* size on a conventional computer since elementary MPP instructions manipulate 128 x 128 bitplanes. Stacks of bitplanes representing integers and 32-bit real numbers are also basic instructions in the MCU which are supported by runtime subroutines in the microprogrammed control unit.

The fundamental data structures for the MPP are shown in Fig. 3.a. The long vector format is supported by the hardware spiral edge connections. However, for long data shifts the vertical interprocessor connections can also be used. The 128 x 128 matrix is the most natural data structure for the MPP. Higher dimensional data structures may also be implemented. If the last two dimensions of the data structure are 128 x 128 then higher dimensions are simply processed serially. If the last two dimensions are less than 128 x 128 then it may be possible to pack more than two dimensions into a single bitplane. For example, a 16 x 8 x 8 x 4 x 4 data structure can be efficiently packed into a 128 x 128 array. Convenient data manipulation routines for this data structure can be efficiently developed at the Parallel Pascal level.

Frequently, the data to be processed by a parallel processor will be in the format of arrays which exceed the fixed range of parallelism of the hardware. Therefore, it is necessary to have special algorithms that will deal with large arrays by breaking them down into blocks manageable by the hardware, without loosing track of the relationships between different blocks.

There are two main schemes for storing large arrays on processor arrays: the *blocked* scheme and the *crinkled* scheme; these are illustrated in Fig. 3.b. Consider that a M x M array is distributed on an N x N processor array where K = M / N is an integer. In the blocked scheme a large array is considered as a set of K x K blocks of size N x N each of which is distributed on the processor. Therefore, elements which ar allocated to a single PE are some multiple of N apart on the large array. In the crinkled scheme each PE contains a K x K matrix of adjacent elements of the large array. Therefore, each parallel processor array contains a sampled version of the large array. For conventional array operations which involve large array shift and rotate operations both blocked and crinkled schemes can be implemented with only a very small amount of overhead. The crinkled scheme is slightly more efficient when shift distances are very small and the blocked scheme has a slight advantage when the shift distance is of the order of N.

The third type of data structure which can be manipulated on the MPP is the huge array which is much too large to fit into the 2 Mb MPP local storage. This scheme, the *spooled* organization, involves the staging memory and is illustrated in Fig 3.c. In the spooled scheme the data is stored in the staging memory and is processed one block at a time by the processor array. The I/O operations to the staging memory are overlapped with data processing so that if the computation applied to each block is large enough then the cost of spooling will be negligible. However, if only a few operations are applied to each block the the I/O time will dominate. For near neighbor operations one possibility is to perform a sequence of operations on each block without regard to other blocks. The boundary elements of the result array will not be valid. This is circumvented by reading overlapping blocks and only writing the valid portions of the result blocks back to memory.

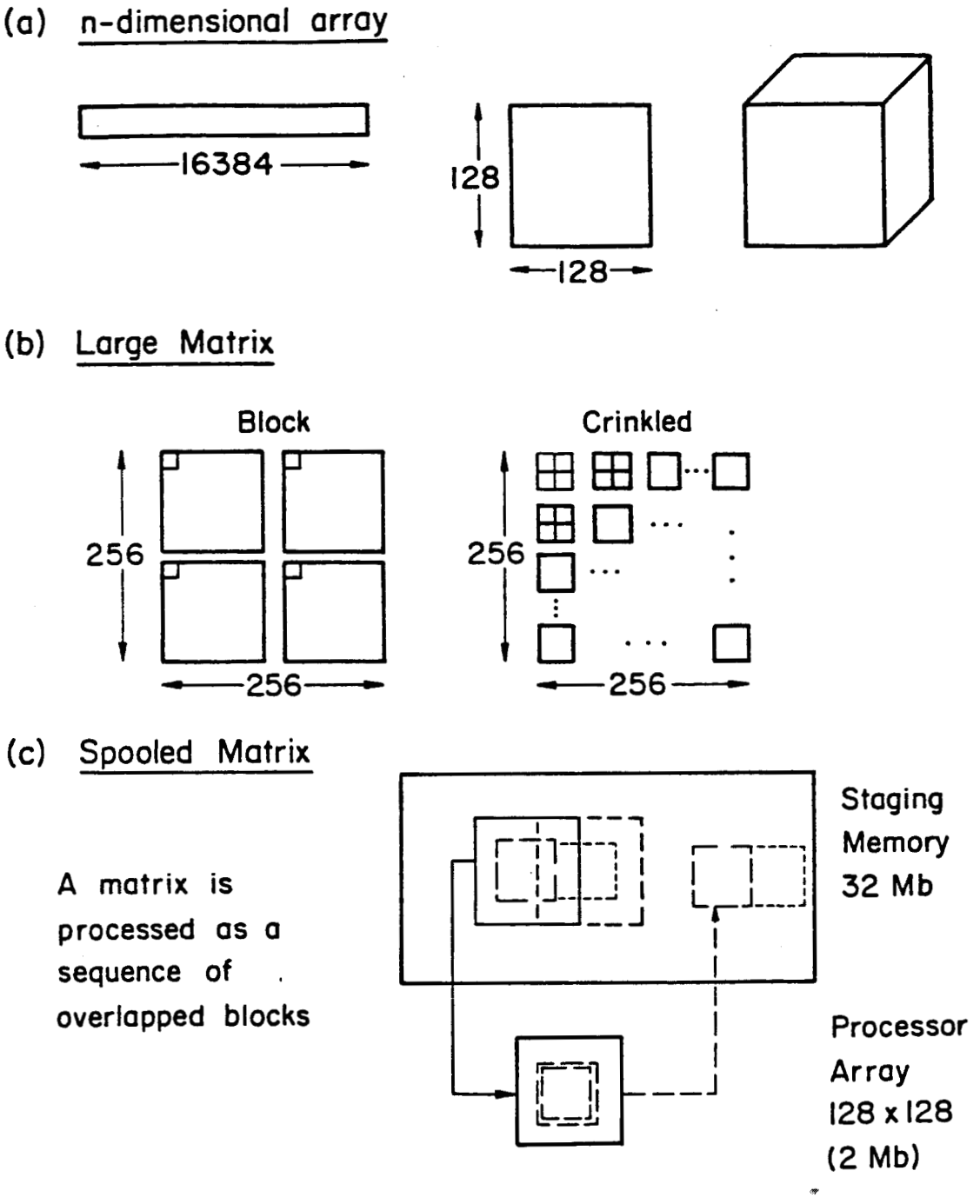


Figure 3. Data Structures for the MPP

THE MPP PROGRAMMING ENVIRONMENT

There are three fundamental classes of operations on array data which are frequently implemented as primitives on array computers but which are not available in conventional programming languages, these are: data reduction, data permutation and data broadcast. These operations have been included as primitives in the high level language for the MPP called Parallel Pascal. Mechanisms for the selection of subarrays and for selective operations on a subset of elements are also important language features.

High level programming languages for mesh connected SIMD computers usually have operations similar to matrix algebra primitives since entire arrays are manipulated with each machine instruction. A description of Parallel Pascal features is given elsewhere in this proceedings [7]. A brief synopsis of important features of this language follows.

Parallel Pascal is an extended version of the Pascal programming language which is designed for the convenient and efficient programming of parallel computers. It is the first high level programming language to be implemented on the MPP. Parallel Pascal was designed with the MPP as the initial target architecture; however, it is also suitable for a large range of other parallel processors. A more detailed discussion of the language design is given in [3].

In Parallel Pascal all conventional expressions are extended to array data types. In a parallel expression all operations must have conformable array arguments. A scalar is considered to be conformable to any type compatible array and is conceptually converted to a conformable array with all elements having the scalar value.

In many highly parallel computers including the MPP there are at least two different primary memory systems; one in the host and one in the processor array. Parallel Pascal provides the reserved word *parallel* to allow programmers to specify the memory in which an array should reside.

Reduction Functions

Array reduction operations are achieved with a set of standard functions in Parallel Pascal. The first argument of a reduction function specifies the array to be reduced and the following arguments specify which dimensions are to be reduced. The numeric reduction functions maximum, minimum, sum and product and the Boolean reduction functions any and all are implemented.

Permutation Functions

One of the most important features of a parallel programming language is the facility to specify parallel array data permutations. In Parallel Pascal three such operations are available as primitive standard functions: *shift*, *rotate* and *transpose*.

The shift and rotate primitives are found in many parallel hardware architectures and also, in many algorithms. The shift function shifts data by the amount specified for each dimension and shifts zeros (null elements) in at the edges of the array. Elements shifted out of the array are discarded. The rotate function is similar to the shift function except that data shifted out of the array is inserted at the opposite edge so that no data is lost. The first argument to the shift and rotate functions is the array to be shifted; then there is an ordered set of parameters, each one specifies the amount of shift in its corresponding dimension.

While transpose is not a simple function to implement with many parallel architectures, a significant number of matrix algorithms involve this function; therefore, it has been made available as a primitive function in Parallel Pascal. The first parameter to transpose is the array to be transposed and the following two parameters specify which dimensions are to be interchanged. If only one dimension is specified then the array is flipped about that dimension.

Distribution Functions

The distribution of scalars to arrays is done implicitly in parallel expressions. To distribute an array to a larger number of dimensions the *expand* standard function is available. This function increases the rank of an array by one by repeating the contents of the array along a new dimension. The first parameter of expand specifies the array to be expanded, the second parameter specifies the number of the new dimension and the last parameter specifies the range of the new dimension.

This function is used to maintain a higher degree of parallelism in a parallel statement which may result in a clearer expression of the operation and a more direct parallel implementation. In a conventional serial environment such a function would simply waste space. For example, to distribute a N-element vector A over all rows of a N x N matrix, the expression is "expand(A,1,1..N)"; as an alternative, to distribute the vector over the columns, the second argument to expand should be changed to 2.

Sub-Array Selection

Selection of a portion of an array by selecting either a single index value or all index values for each dimension is frequently used in many parallel algorithms; e.g., to select the ith row of a matrix which is a vector. In Parallel Pascal all index values can be specified by eliding the index value for that dimension.

Conditional Execution

An important feature of any parallel programming language is the ability to have an operation operate on a subset of the elements of an array. In Parallel Pascal a *where - do - otherwise* programming construct is available which is similar to the conventional *if - then - else* statement except that the control expression results in a Boolean array rather than a Boolean scalar. All parallel statements enclosed by the where statement must have results which are the same size as the controlling array. Only result elements which correspond to true elements in the controlling array will be modified. Unlike the if statement, both clauses of the where statement are always executed.

System Support

In addition to the MPP Parallel Pascal compiler there is a Parallel Pascal translator and a library preprocessor to aide high level program development. The translator translates a Parallel Pascal program into a standard Pascal form. In this way, conventional serial computers can be used to develop and test Parallel Pascal programs if they have a standard Pascal compiler.

Standard Pascal has no library facility; all subprograms i.e., procedures and functions, must be present in the source program. A library preprocessor was developed to allow the use of libraries without violating the rules of standard Pascal.

MPP Compiler Restrictions

The Parallel Pascal compiler for the MPP currently has several restrictions. The most important of these is that the range of the last two dimensions of a parallel array are constrained to be 128; i.e., to exactly fit the parallel array size of the MPP. It is possible that language support could have been provided to mask the hardware details of the MPP array size from the programmer; however, this would be very difficult to do and efficient code generation for arbitrary sized arrays could not be guaranteed. Matrices which are smaller than 128 x 128 can usually be fit into a 128 x 128 array by the programmer. Frequently, arrays which are larger than 128 x 128 are required and these are usually fit into arrays which have a conceptual size which is a multiple of 128 x 128. For example, a large matrix of dimensions (m * 128) x (n * 128) is specified by a four dimensional array which has the dimensions m x n x 128 x 128. There are two fundamental methods for packing the large matrix data into this four dimensional array (see Fig. 3.b), this packing may be directly achieved by the staging memory in both cases.

Host programs for the MPP can be run either on the main control unit (MCU) or on the VAX; in the latter case the MCU simply relays commands from the VAX to the PE array. The advantages of running on the VAX is a good programming environment, floating point arithmetic support and large memory (or virtual memory). The advantage of running on the MCU is more direct control of the MPP array.

Compiler directives are used to specify if the generated code should run on the MCU or the VAX. With the current implementation of the code generator, only complete procedures can be assigned to the MCU and only programs on the MCU can manipulate parallel arrays. Therefore, the programmer must isolate sections of code which deal with the PE array in procedures which are directed to the MCU.

MPP PERFORMANCE EVALUATION

The peak arithmetic performance of the MPP is in the order of 400 million floating point operations per second (MFLOPS) for 32 bit data and 3000 million operations per second (MOPS) for 8-bit integer data. In order to sustain this performance the data matrices to be processed must be as large as the processor array or larger and the amount of time transferring data between processors should be relatively small compared to the time spent on arithmetic computations. For image processing the former constraint is rarely a problem; however, the latter constraint requires careful study.

In order to analyze the effectiveness of the interconnection network for different manipulations it is necessary to characterize the processing speed of the PE and the speed of the interconnection network. On the MPP both of these are data dependent; we have considered three representative cases: single-bit *Boolean* data, 8-bit *integer* data and 32-bit floating-point (*real*) data. For each of these data types we have estimated a *typical* time for an elemental operation. These estimates are of a reasonable order for this minimal PE architecture but are not very precise. For example, the instruction cycle time for a memory access and operation on the MPP is 100 ns. An elemental boolean operation may be considered to take 100 ns.; however, it may be argued that an operation should involve two operands and have all variables in memory in which case three memory accesses (instructions) would require 300ns. For our analysis a two instruction (200 ns) model was used to represent Boolean instruction times. For the real and integer data a convenient number midway between the times for addition and multiplication was used; this was 5 μs. for an integer operation and 40 μs. for a real operation. It should be remembered that elemental operations also include many other functions such as transcendental functions since these can be computed in times comparable to a multiplication on a bit-serial architecture. By adding a large amount of additional hardware to each PE it is possible to increase the speed of multiplication by 10 times or more [8].

For each of the data manipulations considered, times for the three different data types was computed. The performance of the MPP for each manipulation is indicated by the ratio of the data transfer time to an elemental PE operation on the same data type; this will be called the *transfer ratio*. One way to look at this ratio is the number of elemental data operations which must be performed between data transfers for the data transfers not to be the dominant cost for the algorithm. On the MPP data may be shifted between adjacent PE's in one instruction time (100 ns.) concurrently with a PE processing instruction.

Shift and Rotate Operations

The only permutation function which is directly implemented by the MPP is the near neighbor rotate (or shift). The direction of the rotation may be in any of the four cardinal directions. The rotation utilizes the toroidal end around edge connections of the mesh. The *shift* function is similar except that the mesh is not toroidally connected and zeroes are shifted into elements at the edge of the array; therefore, the shift function is not a permutation function in the strict sense. The concept of the rotate and shift functions extends to n dimensions; on the MPP the last two dimensions of the array correspond to the parallel hardware dimensions and are executed in parallel, higher dimension operations are implemented in serial. The cost of the rotate function is dependent on the distance rotated. It also depends on the size of the data elements to be permuted.

The transfer ratios for the shift operation are given in Table 1. Ratios are given for shift distances of 1 and 64 elements; 64 is the largest shift which will normally be required in a single dimension on a 128 x 128 matrix since a shift of 65 can be obtained with a rotate of -63 and a mask operation. The worst case figures for a two dimensional shift is 64 in each direction; i.e., twice the figures given in Table 1.

For single element shifts the interconnection network is more than adequate for all data types. For maximum distance shifts the ratio of 33 for Boolean data could cause problems for some algorithms but the situation is much better for real data.

Table 1: The Cost for Shift and Rotate Operations

Shift distance	Operation cost in μs.			Transfer Ratio		
	Boolean	integer	real	Boolean	integer	real
1	0.2	1.6	6.4	1.0	0.32	0.16
64	6.5	51	210	33	10	5.2

Important Data Manipulations

A simple algorithm to perform any arbitrary data mapping on the MPP is as follows. Start with the address of where the data is to come from in each PE. For each PE compute the distance that the data must be moved to reach that PE. Using the spiral interconnections, rotate the data 16384 times. After each rotation compare the distance in each PE with the distance moved and if they match then store the data for that PE. The transfer ratios for this algorithm are 82000, 10000 and 4200 for Boolean integer and real data types respectively. Obviously, this is much too slow for most practical applications. Fortunately, for most applications only a small number of regular data mappings are required; efficient algorithms can be developed for most of these mappings.

The transfer ratio for a number of important data manipulations is shown in Table 2. The figures for large arrays correspond to the blocked data scheme. For large arrays the transfer ratio is normalized by the cost of an operation on the whole array.

This technique may be applied to almost any parallel architecture. It is expected that the results obtained for the MPP would be very similar to those obtained for other like architectures such as the Distributed Array Processor (DAP) [9] or NCR's GAPP processor chip which contains 72 PE's with local memory [4]. The results given in Table 2 could be used by programmers to predict the performance of algorithms on the MPP. A more detailed analysis of data mappings on the MPP is given in [10].

For the MPP, the results indicate that, although arbitrary data mappings may be very costly, some important data manipulations can be done very efficiently. The shift register, which has a 2 times speedup factor for multi-bit arithmetic also has a significant effect on the implementation of several of the multi-bit data manipulations studied. Especially interesting is the improvement of over 10 times for real data distribution. The shuffle cannot be implemented fast enough for efficient FFT implementation; however, other data mapping strategies for the FFT, such as butterfly permutations, are well known which have a much more efficient implementation on the MPP.

On the DAP row and column distribution is implemented directly by special hardware buses. For the MPP we can see from Table 2 that no advantage would be gained from this hardware for real data operations and possibly very little advantage for integer operations.

The MPP can effectively implement algorithms on pyramid data structures. Horizontal operations are done with near neighbor operations; i.e. single element shifts. Vertical operations require data mappings similar to the shuffle permutation; the times for these operations are given in Table 2. These times for numeric data are quite reasonable for many pyramid algorithms. A detailed analysis of pyramid operations on the MPP is given in [11].

Sorting has been proposed as one technique for doing arbitrary data permutations on the MPP; the cost of bitonic sorting on the MPP is given in Table 2. This cost is very high although not as high as the arbitrary data mapping algorithm.

The last row in Table 2. shows the transfer ratio for swapping a matrix with the staging memory. In this case the transfer ratio indicates the number of operations which

Table 2: Transfer Ratios for Different Data Manipulations and Array Sizes

Data Manipulation	Array Size								
	128 x 128			256 x 256			512 x 512		
	Boolean	integer	real	Boolean	integer	real	Boolean	integer	real
Data Shift									
a) 1 element	1.0	0.32	0.16	2.0	0.5	0.24	2.0	0.5	0.24
b) worst case	33	10.2	5.2	33	10	5.2	33	10	5.2
Broadcast									
a) Global	2	0.64	0.32	0.88	0.28	0.14	0.59	0.20	0.09
c) Row (or column)	68	3.2	0.52	35	1.68	0.30	18.2	0.92	0.19
Shuffle (2-dimensional)	640	90	42	640	90	42	640	90	42
Transpose	840	110	44	840	110	44	840	110	44
Flip	190	43	21	190	43	21	190	43	21
Pyramid Up (sum reduce)	330	45	21	110	15	7.5	28	4	2.3
Pyramid down	330	44	19	110	15	6.4	28	3.9	1.7
Sort	19000	1100	280	12000	870	230	14000	790	210
Swap	256	20	10	256	20	10	256	20	10

must be performed on each swapped matrix for there to be no significant overhead due to swapping. Since the transfer ratio does not change with array size, this suggests that spooling with large matrices for near neighbor operations would be more efficient than spooling with 128 x 128 blocks.

APPLICATIONS

The MPP was originally designed for processing multispectral satellite imagery and synthetic aperture radar imagery. Both of these applications have now been demonstrated on the MPP. Other uses of the MPP are now being explored. There are now 36 active projects on the MPP; these can grouped as follows: physics (10), earth sciences (5) signal and image processing (7), and computer science (14). A full list of these projects is given in Appendix A.

CONCLUSION

The Massively Parallel Processor is a highly effective computer for a large range of scientific applications. It is representative of the class of highly parallel SIMD mesh connected computers. Novel features of the MPP design are the staging memory and the PE shift register. The MPP has demonstrated its capability to implement the image processing algorithms for which it was originally designed; however, the current system lacks the very high speed peripheral devices needed to optimize its performance. It is also being used for a much broader range of scientific applications. The main limitation with the MPP when used for other applications is the limited amount of local memory (1024 bits/PE). This should not be a problem with future systems especially since the recent advances in memory technology. This problem has been offset on the MPP by judicious use of the staging memory.

The problem frequently cited for mesh connected SIMD architectures is the inefficiency of the mesh interconnection scheme when used for other than near neighbor tasks. However, this is not a problem for many practical applications on the MPP; for example, FFT and pyramid operations can be effectively implemented especially for very large data structures.

The MPP is more cost effective for suitable applications than supercomputers of a similar age. Future SIMD mesh connected computers may be anticipated which will take advantage of recent VLSI technology and will be much more powerful than the MPP. These systems can be expected to be much more cost effective than more conventional supercomputers for suitable applications such as low level image processing. These architectures can also be effectively implemented at smaller scales; for example, as attached processors to microprocessor systems.

The cost of using a highly parallel computer is the change of programming style and the need to reformulate existing programs. However, programming in an appropriate high level language is often not conceptually more difficult than programming a conventional computer. In fact, in some respects, it is simpler since arrays are manipulated without the multiple do loops required in conventional serial programming languages.

REFERENCES

1. K. E. Batcher, "Design of a Massively Parallel Processor," *IEEE Transactions on Computers* C-29(9) pp. 836-840 (September 1981).
2. J. L. Potter, *The Massively Parallel Processor*, MIT Press (1985).
3. A. P. Reeves, "Parallel Pascal: An Extended Pascal for Parallel Computers," *Journal of Parallel and Distributed Computing* 1 pp. 64-80 (1984).
4. NCR Corporation, *Geometric Arithmetic Parallel Processor*, NCR, Dayton, Ohio (1984).
5. S. G. Morton, E. Abreau, and F. Tse, "ITT CAP-Toward a Personal Supercomputer," *IEEE Micro*, pp. 37-49 (December 1985).
6. A. P. Reeves, "Fault Tolerance in Highly Parallel Mesh Connected Processors," in *Computing Structures for Image Processing*, ed. M. J. B. Duff, Academic Press (1983).
7. A. P. Reeves, "Languages for Parallel Processors," *International Workshsop on Data Analaysis in Astronomy*, , Erice, Italy(April 1986).
8. A. P. Reeves, "The Anatomy of VLSI Binary Array Processors," in *Languages and Architectures for Image Processing*, ed. M. J. B. Duff and S. Levialdi, Academic Press (1981).
9. R. W. Gostick, "Software and Algorithms for the Distributed-Array Processor," *ICL Technical Journal*, pp. 116-135 (May 1979).
10. A. P. Reeves and C. H. Moura, "Data Manipulations on the Massively Parallel Processor," *Proceedings of the Nineteenth Hawaii International Conference on System Sciences*, pp. 222-229 (January, 1986).
11. A. P. Reeves, "Pyramid Algorithms on Processor Arrays," *Proceedings of the NATO Advanced Research Workshop on Pyramidal Systems for Image Processing and Computer Vision*, , Maratea, Italy(May 1986).

APPENDIX A. Current Research Using the MPP

Dr. John A. Barnden Indiana University	Diagramtic Information-Processing in Neural Arrays
Dr. Richard S. Bucy Univ. of Southern California	Fixed Point Optimal Nonlinear Phase Demodulation
Dr. Gregory R. Carmichael University of Iowa	Tropospheric Trace Gas Modeling on the MPP

Dr. Tara Prasad Das State University of New York at Albany	Investigations on Electronic Structures and Associated Hyperfine Systems Using the MPP
Dr. Edward W. Davis North Carolina State University	Graphic Applications of the MPP
Dr. Howard B. Demuth University of Idaho	Sorting and Signal Processing Algorithms: A Comparison of Parallel Architectures
Dr. James A. Earl University of Maryland	Numerical Calculations of Charges Particle Transport
Mr. Eugene W. Greenstadt TRW	Space Plasma Graphics Animation
Dr. Chester E. Grosch NASA-Langley Research Center	Adapting a Navier-Stokes Code to the MPP
Dr. Robert J. Gurney Goddard Space Flight Center	A Physically-Based Numerical Hillslope Hydrological Model with Remote Sensing Calibration
Dr. Martin Hagan University of Tulsa	Sorting and Signal Processing Algorithms: A Comparison of Parallel Architectures
Dr. Harold M. Hastings Hofstra University	Applications of Stochastic and Reaction – Diffusion Cellular Automata
Dr. Sara Ridgway Heap Goddard Space Flight Center	Automatic Detection and Classification of Galaxies on "Deep-Sky" Pictures
Dr. Nathan Ida The University of Akron	Solution of Complex, Linear Systems of Equations
Dr. Robert V. Kenyon MIT Man Vehicle Laboratory	Application of Parallel Computers to Biomedical Image Analysis
Dr. Daniel A. Klinglesmith, III Goddard Space Flight Center	Comet Haley Large-Scale Image Analysis
Dr. Daniel A. Klinglesmith, III Goddard Space Flight Center	FORTH, an Interactive Language for Controlling the MPP
Dr. Chin S. Lin Southwest Research Institute	Simulation of Beam Plasma Interactions Utilizing the MPP
Dr. Stephen A. Mango Naval Research Laboratory	Synthetic Aperture Radar Processor System Improvements
Dr. Michael A. McAnulty University of Alabama	Algorithmic Commonalities in The Parallel Environment
Dr. A.P. Mulhaupt University of New Mexico	Kalman Filtering and Boolean Delay Equations on an MPP

Dr. John T. O'Donnell Indiana University	Simulating an Applicative Programming Storage Architecture Using the NASA MPP
Mr. Martin Ozga USDA-Statistical Reporting Service (SRS)	A Comparison of the MPP with Other Supercomputers for Landsat Data Processing
Dr. H.K. Ramapriyan Goddard Space Flight Center	Development of Automatic Techniques for Detection of Geological Fracture Patterns
Dr. John Reif Harvard University Computer Science	Parallel Solution of Very Large Sparse Linear Systems
Dr. L.R. Owen Storey Stanford University	Particle Simulation of Plasmas on the MPP
Dr. James P. Strong Goddard Space Flight Center	Development of Improved Techniques for Generating Topographic Maps from Spacecraft Imagery
Dr. Francis Sullivan National Bureau of Standards	Phase Separation by Ising Spin Simulations
Dr. Peter Suranyi University of Cincinnati	A Study of Phase Transitions in Lattice Field Theories on the MPP
Dr. James C. Tilton Goddard Space Flight Center	Use of Spatial Information for Accurate Information Extraction
Dr. William Tobocman Case Western Reserve University	Wave Scattering by Arbitrarily Shaped Targets Direct and Inverse
Mr. Lloyd A. Treinish Goddard Space Flight Center	Animated Computer Graphics Models of Space and Earth Sciences Data Generated via the MPP
Dr. Scott Von Laven KMS Fusion, Inc.	Free-Electron Laser Simulations on the MPP
Dr. Elden C. Whipple, Jr. UCSD/CASS/C-001	A Magnetospheric Interactive Model Incorporating Current Sheets (MIMICS)
Dr. Richard L. White Space Telescope Science Institute	The Dynamics of Collisionless Stellar Systems
Dr. Lo I. Yin Goddard Space Flight Center	Reconstruction of Coded-Aperature X-ray Images

Appendix E

Proceedings of the Nato Advanced Research Workshop on Pyramidal Systems for Image Processing and Computer Vision Maratea, Italy, May 1986

PYRAMID ALGORITHMS ON PROCESSOR ARRAYS

Anthony P. Reeves
School of Electrical Engineering
Cornell University
Ithaca, New York 14853

Abstract

A class of adaptive grid size algorithms, called pyramid algorithms, have received much attention in recent years for computer vision applications. Special hardware architectures which are optimal for grid resolution transformations have been proposed to implement these algorithms. In this chapter analysis techniques are described which measure the effectiveness of different architectures. A comparison is made between the more conventional planar or flat architecture and the optimal pyramid architecture. It is shown that in many cases the additional hardware of the pyramid scheme offers little improvement in performance.

INTRODUCTION

The general concept of pyramid algorithms is to decompose a large matrix into a number of lower resolution matrices for more convenient processing. The most typical pyramid structure consists of a square base matrix with dimensions N x N (where N is a power of 2) and $\log_2$ N reduced matrices (layers) each layer having dimensions half that of the layer below until a single element apex (layer 0) is reached. A number of algorithms for image processing have been based on this data structure. Other applications such as multigrid techniques for PDE's can also be considered in the pyramid framework.

In order to achieve the very high data processing rates needed for image processing applications a number of highly parallel computer architectures have been designed and implemented; several of such systems are now commercially available. A very effective parallel computer architecture for conventional image processing applications is the mesh connected processor array. For processing pyramid data structures a number of researchers have proposed hardware modifications to the two dimensional processor array. These modifications directly match the pyramid data structure by having interconnected layers of processor arrays attached to the base processor array. In this paper techniques are developed to investigate the efficiency with which pyramid algorithms can be implemented on processor arrays and the effectiveness of enhanced hardware schemes is considered.

A major consideration in the design of a highly parallel computer architecture is the processor interconnection network. Processors must communicate with a speed which does not impede data processing; however, a general processor interconnection network is usually prohibitively expensive when a large number of processors are involved. A major design task is to design a restricted network which is adequate for the anticipated tasks for the system. The mesh interconnection scheme is simple to implement and is well suited to a large number of image processing algorithms. The first section of this paper describes the Massively Parallel Processor (MPP) [1] and the performance of the processor interconnection network is considered in detail.

The second section of this paper deals with the implementation of pyramid algorithms on the MPP and also considers the effect of adding a pyramid hardware structure to the MPP. A two dimensional processor array will be called a *flat* processor and a processor array with additional hardware for pyramid data structures will be called a *pyramid* processor. The general techniques used to analyze these architectures may be used to determine the effectiveness of other highly parallel processor systems for pyramid applications.

The hardware enhancement of a pyramid of processing arrays approaches $\frac{4}{3}$ times the number of processing elements (PE's) required for the base array, as the number of layers increases. Therefore, the maximum possible increase in arithmetic capability is $\frac{4}{3}$. However, the major advantage claimed for the pyramid processor is the speedup in interprocessor data routing. In order to compute a global function over an N x N processor array N processing steps are required. However, on a pyramid processor global information may be extracted at the apex of the pyramid after $\log_2 N$ steps. The potential gain for the pyramid system is therefore $N/\log_2 N$; this value is shown in Table 1. for different values of N. The MPP has a 128 x128 base array therefore the potential speedup is in the order of 18.

The pyramid computer has an improved performance for two types of operations: (a) multiresolution algorithms which involve frequent changes between matrix resolutions and (b) global feature extraction operations. However, the additional pyramid hardware must be justified by the multiresolution operations alone since there are more cost effective hardware enhancements for global feature extraction [2].

Table 1: Data routing advantage of the Pyramid Architecture

N	4	8	16	32	64	128	256	512	1024
$N/\log_2 N$	2	2.33	4	6.4	10.66	18.29	32	56.8	102.4

The cost of the additional hardware for the pyramid processor has been considered by some to be simply $\frac{4}{3}$ times the cost of the flat processor since this is the increase in the number of PE's. The cost will, in general, be much higher than this since each PE now requires more interconnections (an increase from 4 to 9 in our example) and interprocessor connections are no longer local which could be particularly expensive for multiprocessor chip designs.

FLAT PROCESSOR ARRAYS

The Massively Parallel Processor consists of 16384 bit-serial Processing Elements (PE's) connected in 128 x 128 mesh [1]. That is each PE is connected to its 4 adjacent neighbors in a planar matrix. The two dimensional grid is one of the simplest interconnection topologies to implement, since the PE's themselves are set out in a planar grid fashion and all interconnections are between adjacent components. Furthermore, this topology is ideal for two dimensional filtering operations which are common to low level image processing such as small window convolution.

The PE's are bit-serial, i.e. the data paths are all one bit wide. This organization offers the maximum flexibility, at the expense of the highest degree of parallelism, with the minimum number of control lines. For example, as an alternative to the MPP consider 2048 8-bit wide PE's (on the MPP one chip contains 8 1-bit PE's). The 8-bit version would have a less rich set of instructions restricted to predefined byte operations while the bit-serial processors can process any data format. The advantage gained with the 8-bit system is that full processor utilization is achieved with arrays of 2048 elements while arrays of 16384 elements are required for full utilization of the MPP. The MPP PE is well matched to low level image processing tasks which often involve very large data arrays of short integers which may be from 1 to 16 bits.

The effectiveness of the MPP architecture for various interprocessor data manipulations is considered. The MPP offers a simple basic model for analysis since it involves just mesh interconnections and bit-serial PE's. The minimal architecture of the MPP is of particular interest to study, since any architecture modifications to improve performance would result in a more complex PE or a more dense interconnection strategy. The MPP is programmed in a high level language called Parallel Pascal [3].

The MPP Processing Element

The MPP processing element is shown in Fig. 1. All data paths are one bit wide and there are 8 PE's on a single CMOS chip with the local memory on external memory chips.

Except for the shift register, the design is essentially a minimal architecture of this type. The single bit full adder is used for arithmetic operations and the Boolean processor, which implements all 16 possible two input logical functions, is used for all other operations. The NN select unit is the interface to the interprocessor network and is used to select a value from one of the four adjacent PE's in the mesh.

The S register is used for I/O. A bit plane is slid into the S registers independent of the PE processing operation and it is then loaded into the local memory by cycle stealing one cycle. The G register is used in masked operations. When masking is enabled only PE's in which the G register is set perform any operations; the remainder are idle. The masked operation is a very common control feature in SIMD designs. Not shown in Fig. 1. is an OR bus

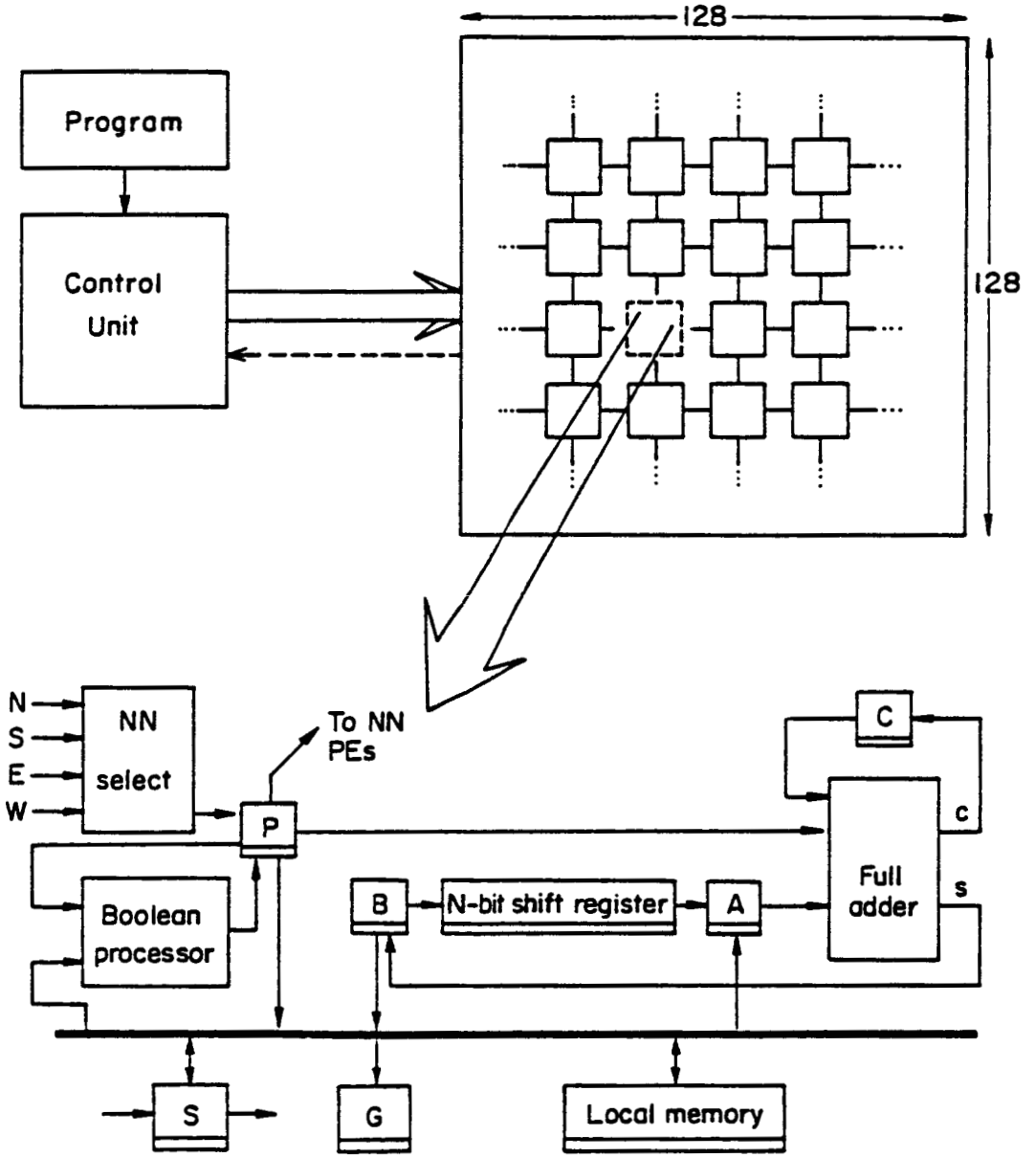


Figure 1. The MPP Processing Element

output from the PE. All these outputs are connected (ORed) together so that the control unit can determine if any bits are set in a bitplane in a single instruction. On the MPP the local memory has 1024 words (bits) and is implemented with bipolar chips which have a 35 ns access time.

The main novel feature of the MPP PE architecture is the reconfigurable shift register. It may be configured under program control to have a length from 2 to 30 bits. Improved performance is achieved by keeping operands circulating in the shift register which greatly reduces the number of local memory accesses and instructions. It speeds up integer multiplication by a factor of two and also has an important effect on floating-point performance.

Flat Array Performance Evaluation

In order to analyze the effectiveness of the interconnection network for different manipulations it is necessary to characterize the processing speed of the PE and the speed of the interconnection network. On the MPP both of these are data dependent; we have considered three representative cases: single-bit *Boolean* data, 8-bit *integer* data and 32-bit floating-point (*real*) data. For each of these data types we have estimated a *typical* time for an elemental operation. These estimates are of a reasonable order for this minimal PE architecture but are not very precise. For example, the instruction cycle time for a memory access and operation on the MPP is 100 ns. An elemental boolean operation may be considered to take 100 ns.; however, it may be argued that an operation should involve two operands and have all variables in memory in which case three memory accesses (instructions) would require 300ns. For our analysis a two instruction (200 ns) model was used to represent Boolean instruction times. For the real and integer data a convenient number midway between the times for addition and multiplication was used; this was 5 μs. for an integer operation and 40 μs. for a real operation. It should be remembered that elemental operations also include many other functions such as transcendental functions since these can be computed in times comparable to a multiplication on a bit-serial architecture. By adding a large amount of additional hardware to each PE it is possible to increase the speed of multiplication by 10 times or more [4].

For each of the data manipulations considered, times for the three different data types was computed. The performance of the MPP for each manipulation is indicated by the ratio of the data transfer time to an elemental PE operation on the same data type; this will be called the *transfer ratio*. One way to look at this ratio is the number of elemental data operations which must be performed between data transfers for the data transfers not to be the dominant cost for the algorithm. On the MPP data may be shifted between adjacent PE's in one instruction time (100 ns.) concurrently with a PE processing instruction.

For many applications the physical dimensions of the parallel hardware are smaller than the dimensions of the array to be processed. In this case the data array is processed as a set of blocks. An extension of the data manipulation algorithms to deal with this situation is discussed.

Shift and Rotate Operations

The only permutation function which is directly implemented by the MPP is the near neighbor rotate (or shift). The direction of the rotation may be in any of the four cardinal directions. The rotation utilizes the toroidal end around edge connections of the mesh. The *shift* function is similar except that the mesh is not toroidally connected and zeroes are shifted into elements at the edge of the array; therefore, the shift function is not a permutation function in the strict sense. The concept of the rotate and shift functions extends to n dimensions; on the MPP the last two dimensions of the array correspond to the parallel hardware dimensions and are executed in parallel, higher dimension operations are implemented in serial. The cost of the rotate function is dependent on the distance rotated. It also depends on the size of the data elements to be permuted.

The transfer ratios for the shift operation are given in Table 2. Ratios are given for shift distances of 1 and 64 elements; 64 is the largest shift which will normally be required in a single dimension on a 128 x 128 matrix since a shift of 65 can be obtained with a rotate of -63 and a mask operation. The worst case figures for a two dimensional shift is 64 in each direction; i.e., twice the figures given in Table 2.

For single element shifts the interconnection network is more than adequate for all data types. For maximum distance shifts the ratio of 33 for Boolean data could cause problems for some algorithms but the situation is much better for real data.

Large Array Processing

Frequently the data to be processed by a parallel processor will be in the format of arrays which exceed the fixed range of parallelism of the hardware. Therefore, it is

Table 2: The Cost for Shift and Rotate Operations

Shift distance	Operation cost in μs.			Transfer Ratio		
	Boolean	integer	real	Boolean	integer	real
1	0.2	1.6	6.4	1.0	0.32	0.16
64	6.5	51	210	33	10	5.2

necessary to have special algorithms that will deal with large arrays by breaking them down into blocks manageable by the hardware, without loosing track of the relationships between different blocks.

There are two main schemes for storing large arrays on processor arrays: the *blocked* scheme and the *crinkled* scheme. Consider that a M x M array is distributed on an N x N processor array where K = M / N is an integer. In the blocked scheme a large array is considered as a set of K x K blocks of size N x N each of which is distributed on the processor. Therefore, elements which ar allocated to a single PE are some multiple of N apart on the large array. In the crinkled scheme each PE contains a K x K matrix of adjacent elements of the large array. Therefore, each parallel processor array contains a sampled version of the large array. For conventional array operations which involve large array shift and rotate operations both blocked and crinkled schemes can be implemented with only a very small amount of overhead. The crinkled scheme is slightly more efficient when shift distances are very small and the blocked scheme has a slight advantage when the shift distance is of the order of N.

The Performance of the MPP

The transfer ratio for a number of important data manipulations is shown in Table 3. The figures for large arrays correspond to the blocked data scheme. For large arrays the transfer ratio is normalized by the cost of an operation on the whole array; i.e., K x K array operations.

This technique may be applied to almost any parallel architecture. It is expected that the results obtained for the MPP would be very similar to those obtained for other like architectures such as the Distributed Array Processor (DAP) [5] or NCR's GAPP processor chip which contains 72 PE's with local memory [6]. The results given in Table 3 could be used by programmers to predict the performance of algorithms on the MPP.

For the MPP, the results indicate that, although arbitrary data mappings may be very costly, some important data manipulations can be done very efficiently. The shift register, which has a 2 times speedup factor for multi-bit arithmetic also has a significant effect on the implementation of several of the multi-bit data manipulations studied. Especially interesting is the improvement of over 10 times for real data distribution. The shuffle cannot be implemented fast enough for efficient FFT implementation; however, other data mapping strategies for the FFT, such as butterfly permutations, are well known which have a much more efficient implementation on the MPP. A more detailed analysis of data mappings on the MPP is given in [7].

On the DAP row and column distribution is implemented directly by special hardware buses. For the MPP we can see from Table 3 that no advantage would be gained from this

hardware for real data operations and possibly very little advantage for integer operations.

PYRAMID PROCESSING

A Pyramid Architecture

A hypothetical architecture of a pyramid machine based on the MPP is illustrated in Fig. 2. The PE is similar to the MPP PE with the addition of five additional interprocessor connections. Four of these are connected to the four children at the next lower level and the fifth is connected to the parent PE in the layer above.

Pyramid Primitive Operations

We consider processing operations on pyramid data structures to be one of three types: *elemental*, in which no communication between PE's is involved, *horizontal*, in which adjacent elements in the same pyramid level are combined, and *vertical*, in which elements at different adjacent levels are combined. Furthermore, we consider that these operations may be applied to the total pyramid or to a subset of levels. Other relevant pyramid operations include feature extraction and data broadcast.

Pyramid Operations on a Flat Array

An efficient scheme for mapping a pyramid data structure to a N x N processor array is as follows. All levels of the pyramid which have dimensions less than N x N are mapped into a single plane. The level with size N x N is mapped into a second plane and levels lower than this are mapped in such a way that each PE is allocated to a sub-pyramid with

Table 3: Transfer Ratios for Different Data Manipulations and Array Sizes

Data Manipulation	Array Size								
	128 x 128			256 x 256			512 x 512		
	Boolean	integer	real	Boolean	integer	real	Boolean	integer	real
Data Shift									
a) 1 element	1.0	0.32	0.16	2.0	0.5	0.24	2.0	0.5	0.24
b) worst case	33	10.2	5.2	33	10	5.2	33	10	5.2
Broadcast									
a) Global	2	0.64	0.32	0.88	0.28	0.14	0.59	0.20	0.09
c) Row (or column)	68	3.2	0.52	35	1.68	0.30	18.2	0.92	0.19
Shuffle (2-dimensional)	640	90	42	640	90	42	640	90	42
Transpose	840	105	44	840	105	44	840	105	44

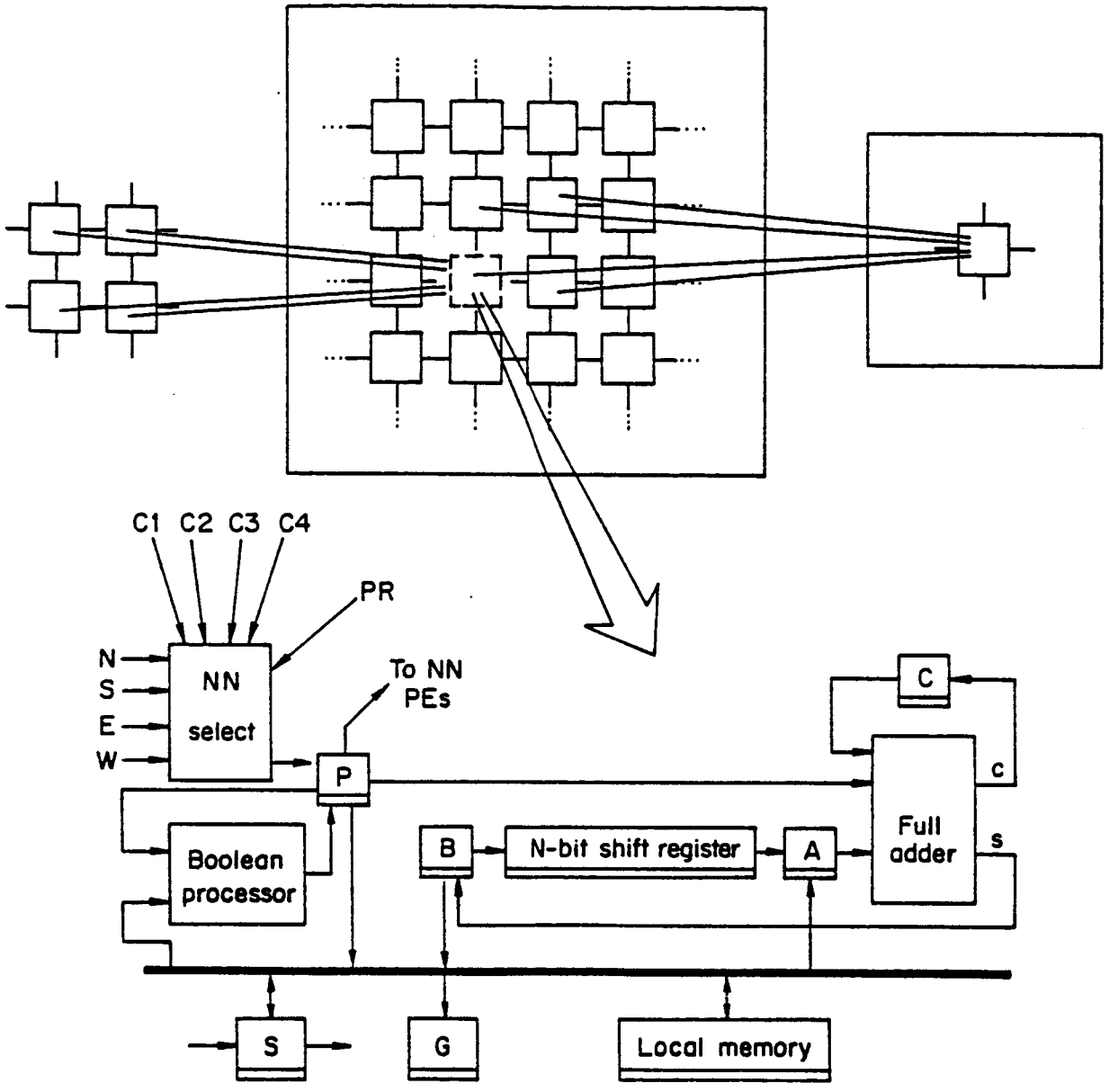


Figure 2. The Pyramid Processing Element

adjacent elements. This is achieved by storing each level with the crinkled storage format. Therefore, for a k-level pyramid the top n - 1 levels ($n = \log_2 N$) are stored in one plane, and the remaining k - n levels are stored in $\sum_{i=0}^{k-n-1} 2^{2i} = \frac{1}{3}(2^{2(k-n)}-1)$ planes. Data transitions between the top n - 1 levels is achieved with perfect shuffle permutations. The perfect shuffle for an 8 x 8 matrix is shown in Fig. 3. Each quadrant of the matrix is distributed over the whole matrix. A 4 x 4 pyramid structure embedded in an 8 x 8 matrix is shown in Fig. 4. With this organization, a transition to a lower level can be achieved by a shuffle and a transition to a higher level can be achieved with an inverse shuffle for all levels simultaneously. This scheme makes good use of memory and is optimal for horizontal operations but is

(a)

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

(b)

1	5	2	6	3	7	4	8
33	37	34	38	35	39	36	40
9	13	10	14	11	15	12	16
41	45	42	46	43	47	44	48
17	21	18	22	19	23	20	24
49	53	50	54	51	55	52	56
25	29	26	30	27	31	28	32
57	61	58	62	59	63	60	64

Figure 3. The Two Dimensional Perfect Shuffle.
The shuffled version of matrix (a) is shown in (b).

relatively slow for vertical operations.

A Comparative Analysis

To evaluate the cost of macro operations a *normalized cost* is used which is similar in concept to the transfer ratio. It is obtained by dividing the actual cost by the cost of an elemental operation on the data structure being processed by the macro operation.

Elemental Operations

The actual time cost in terms of the number of processor array arithmetic operations is shown in Fig. 5 for an elemental operation applied to the whole pyramid data structure and for an architecture with a 128 x 128 base size. The cost for a flat array is shown by the solid

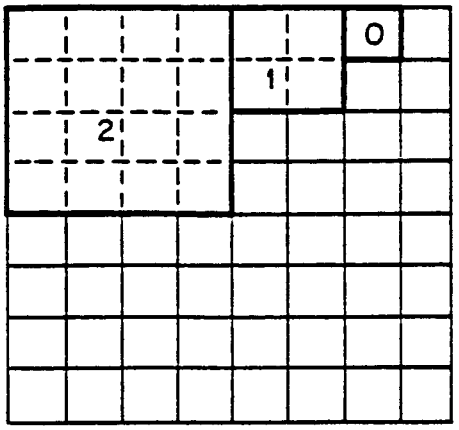


Figure 4. A 4 x 4 base pyramid embedded in an 8 x 8 mesh

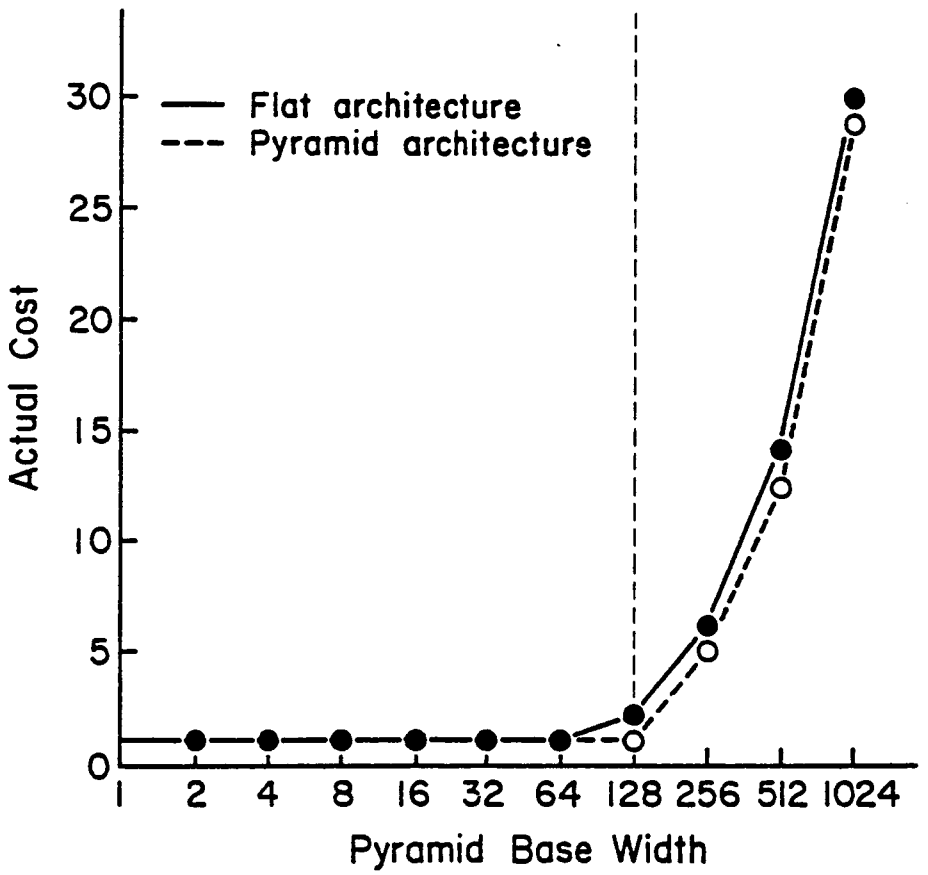


Figure 5. The Actual Cost for Elemental Pyramid Operations

line. While the base of the pyramid to be processed is less than 128 x 128 then the whole pyramid is embedded in a single array and only one operation is required. For a 128 x 128 base pyramid 2 operations are required and a blocked processing strategy is used for larger

pyramids. The cost for the pyramid architecture is shown by the broken line. In this case the 128 x 128 pyramid is exactly matched to the hardware and only requires one operation. Larger pyramids must be blocked using the pyramid base level.

The normalized cost of a elemental operation on the whole pyramid data structure is shown in Fig. 6. Since we normalize by the cost of an operation on the whole pyramid, in this case the result is a constant 1 for all pyramid sizes. For comparison purposes all normalizations are done with respect to an operation on a flat array. The broken line in Fig. 6 shows the normalized cost for an elemental operation for a true pyramid architecture with a 128 x 128 base. When the base of the pyramid to be processed is less than 128 then the cost is the same for both architectures. When the base is 128 x 128 then a 50% reduction in normalized cost for the pyramid architecture is observed. For pyramids with bases larger than 128 x 128 both architectures must process the data in blocks and the advantage of the pyramid architecture rapidly diminishes. It is important to note that the normalized cost for this graph and for all following graphs is plotted on a logarithmic scale.

Large Pyramid Processing

When the pyramid structure ro be processed has a base which is larger than the processor array then a blocking scheme must be used for the lower large levels of the pyramid. An effective scheme is to use the crinkled storage strategy for each large level. In this way each PE contains a complete sub pyramid; therefore, level changes for the large levels are done locally within PE's and no interprocessor communications are necessary. The crinkled

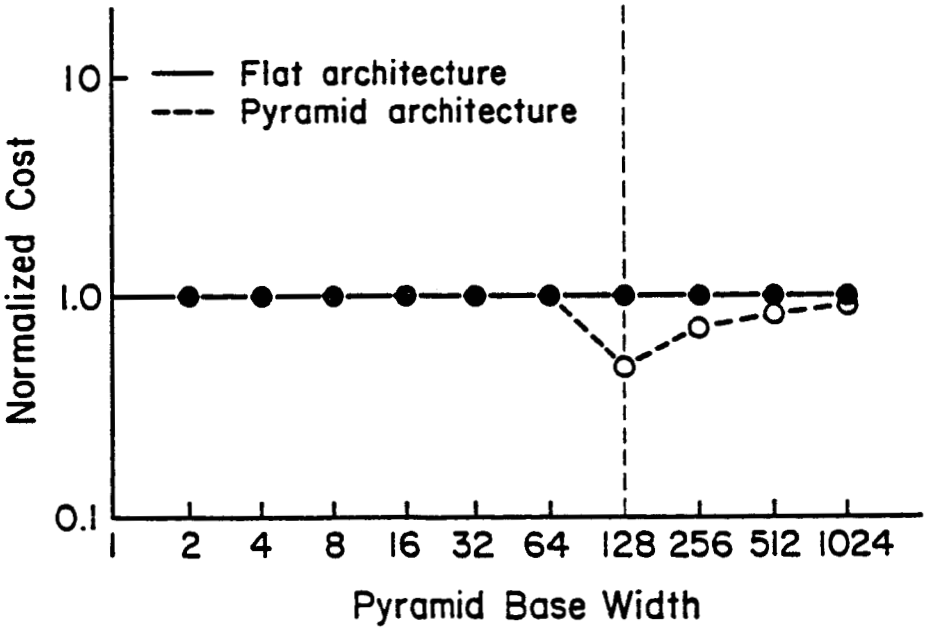


Figure 6. The Normalized Cost for Elemental Pyramid Operations

storage scheme is also very efficient for horizontal operations.

Horizontal Operations

Adjacent element information is readily available for both the architectures considered. Therefore, the normalized cost for macro operations involving horizontal near neighbor information will be similar to Fig. 6 except that the graph will be translated upwards by the time for the macro operation to be implemented in terms of elemental operations. The cost of some important horizontal operations is shown in Fig. 7. For reference, the cost of an add operation is first shown; it is less than one since an add requires less time than a multiply for integer and real data types. A 3 x 3 mean operation is achieved with four near neighbor additions, while a 3 x 3 convolution requires 9 multiplications and 8 near neighbor additions. A symmetric 5 x 5 convolution, such as that used by Burt for Pyramid building [8] requires 6 multiplications and 8 near neighbor additions. An example of an expensive macro operation is the 10 by 10 convolution which requires 100 multiplications and 99 near neighbor additions.

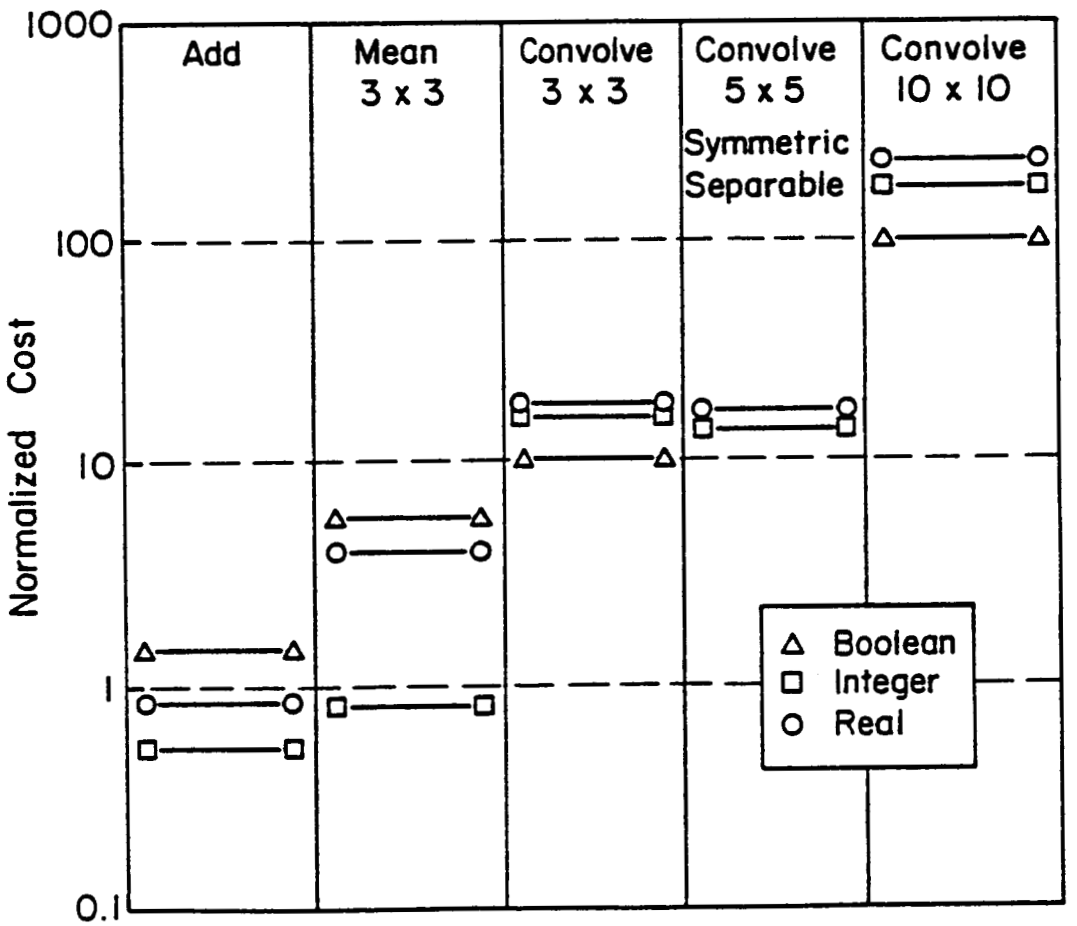


Figure 7. The Normalized Cost of some Horizontal Operations

Vertical Pyramid Operations

It is in the vertical operations that the pyramid architecture is expected to excel in comparison to a flat architecture. An important operation in many pyramid algorithms is to build a pyramid from an image located at the lowest level of the pyramid. We consider a very simple building operation in which each higher level is constructed from the mean of the four sons of the adjacent lower level.

Pyramid Building

The cost of pyramid building is shown for a flat processor in Fig. 8 and for a pyramid processor in Fig. 9. The building cost is normalized by an elemental operation applied to the whole pyramid data structure. We see that for the worst case base width of 128 the flat processor is several times slower that the pyramid however this advantage rapidly diminishes if large pyramids are processed. The Boolean case is particularly bad for the flat array

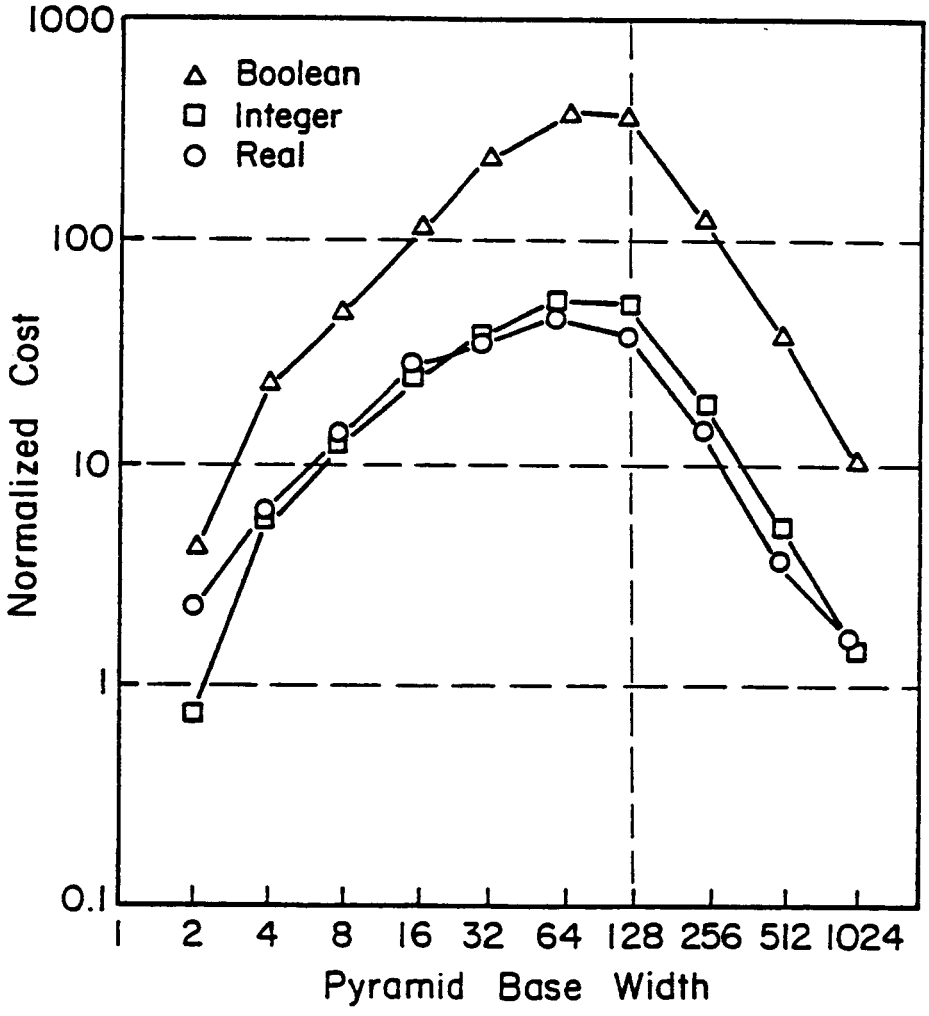


Figure 8. The Normalized Cost for Simple Pyramid Building on a Flat Processor Array

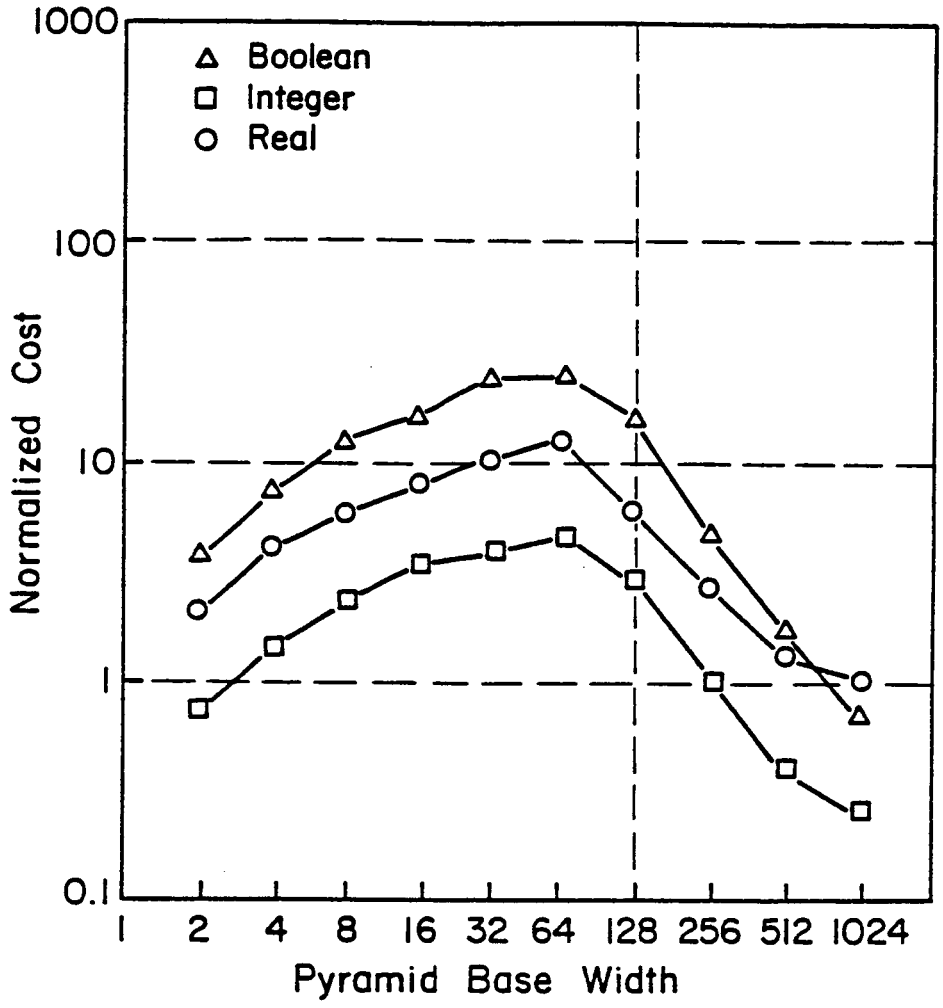


Figure 9. The Normalized Cost for Simple Pyramid Building on a Pyramid Processor Array

because the overhead to do the pyramid data manipulations cannot be amortized over the number of bits in the data as it can for the numeric data formats. The flat array although slower than the pyramid may not be significantly slower in a pyramid building algorithm. For example consider the overlapped algorithm of Burt [8]. At each level a 5 x 5 convolution must be computed before proceeding to the next. The cost of computing these convolutions is comparable to the data building cost.

Another way to look at the vertical operation cost is to consider a single level reduction; this is shown for both flat and pyramid processors in Fig. 10. In this case the cost is normalized by an elemental operation on the level at which the reduced data is generated. The architectures have the same performance when the base is smaller than the level being processed. The worst case for the flat processor occurs for the level which is half the base size since this requires the most interprocessor data transfers. If many operations fall on this level transition then one technique is to consider the processor array to have a size of 64 x 64. The arithmetic processing speed is reduced by four (since only a quarter of the processors are

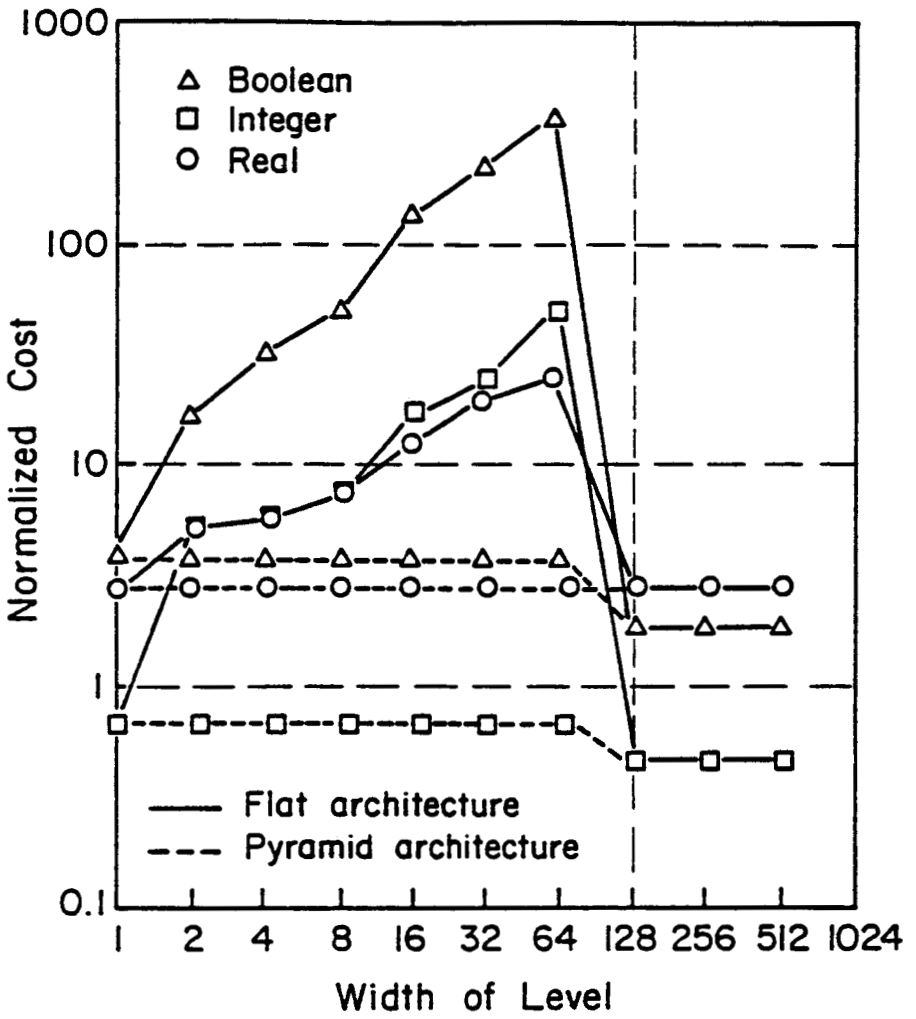


Figure 10. The Normalized Cost for Single Layer Reduction

used) but the data transition is managed at least ten times faster.

Data Reduction

In Fig. 11 and Fig. 12 the cost for computing the global sum of the elements in the base of the pyramid is given. In this case the cost is normalized by an elemental operation over the whole pyramid data structure. The advantage of the pyramid architecture is not as great as in the pyramid building example because the flat array is no longer forced to emulate the pyramid in the highest levels since only the final sum is required.

CONCLUSIONS

A general technique for determining the cost of pyramid algorithms has been presented. This technique may be used on any flat array architecture to evaluate its performance for pyramid algorithms and to evaluate the benefit of a pyramid processor hardware enhancement. Once the graphs similar to those given in Figs. 6-12 have been obtained then the mix

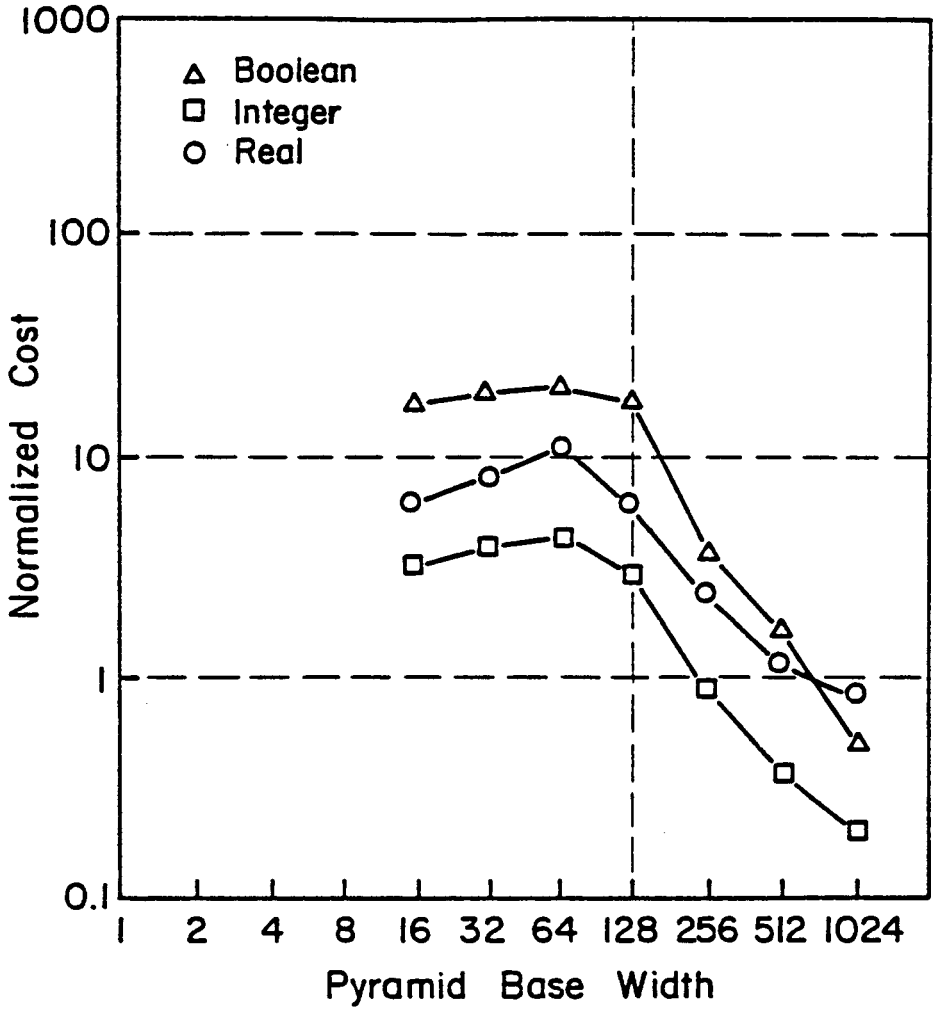


Figure 11. The Normalized Cost for Sum Reduction on a Flat Processor Array

of vertical to other pyramid operations for the desired application must also be determined for an accurate evaluation.

From the results given in this paper the following observations can be made.

1. Pyramid algorithms can be effectively implemented on a flat array in many cases; each application should be examined individually in detail.
2. If most of the operations are performed in the base of the pyramid then the pyramid hardware will offer little advantage.
3. The flat array is substantially worse for Boolean data types than for numeric data types. Therefore, a preponderance of Boolean data structures favors the pyramid approach. However, if numeric data types dominate then the case for pyramid hardware is much weaker.
4. The pyramid hardware can only be justified by the need for a large number of pyramid level changes in a algorithm. It cannot be justified by the global data reduction function alone since other techniques can be used on the flat array.

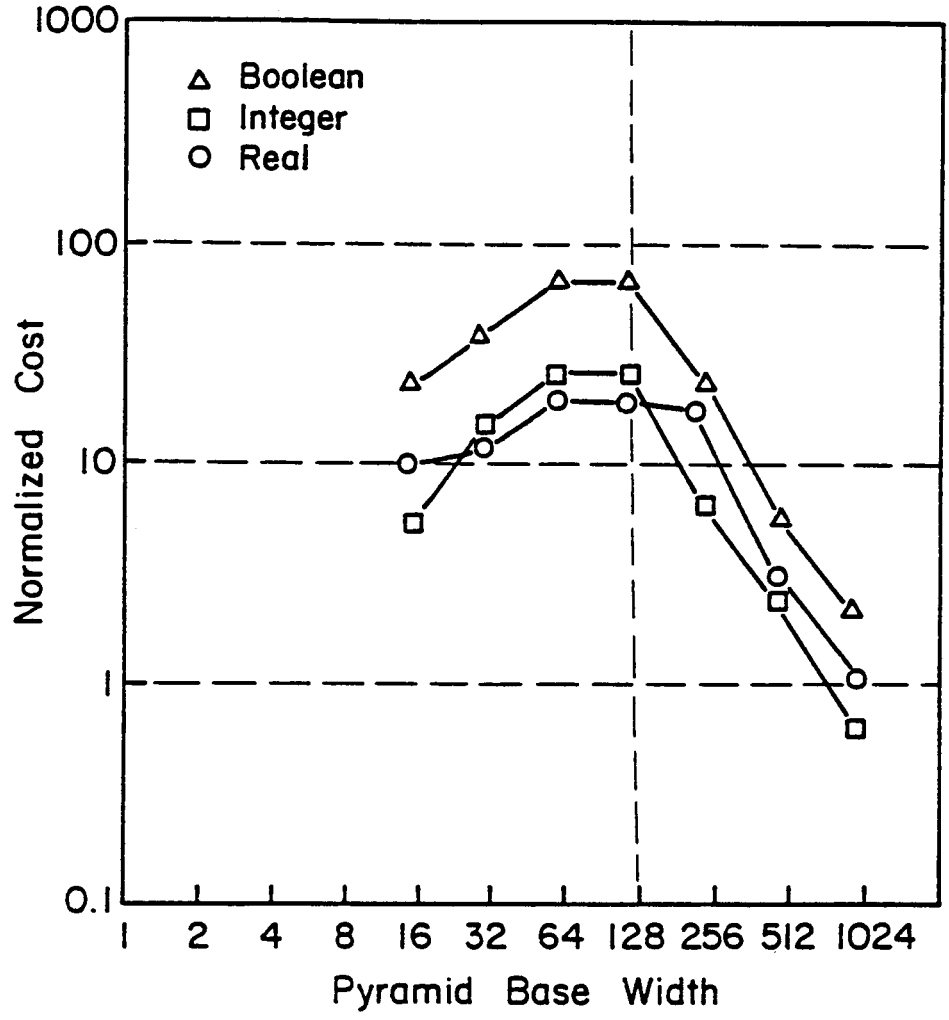


Figure 12. The Normalized Cost for Sum Reduction on a Pyramid Processor Array

5. The ratio of the hardware base size and the pyramid base size will have a major effect on the performance. For numeric data processing one way to increase the effectiveness of the flat array is to use a smaller number of multi-bit processors; therefore the hardware base size is reduced without loss of processing power.

6. When the pyramid to be processed is much larger than the hardware processor array then there is very little advantage in having the pyramid hardware. This is the case when the MPP is used to process pyramids with a base size of 512 x 512.

In summary while the analysis is complex and very application dependent there are a number of situations which may be identified for which justification for the pyramid architecture can be determined. The pyramid hardware is hard to justify if either (a) large pyramid processing techniques are used, (b) the operations to be performed are numeric, or (c) vertical operations do not constitute a large part of the pyramid algorithms to be implemented. The strongest case for the pyramid hardware exists when Boolean data dominates

the processing and the number of PE's exactly matches the number of elements in the pyramid data structure.

REFERENCES

1. K. E. Batcher, "Design of a Massively Parallel Processor," *IEEE Transactions on Computers* C-29(9) pp. 836-840 (September 1981).
2. A. P. Reeves, "On Efficient Global Information Extraction Methods For Parallel Processors," *Computer Graphics and Image Processing* 14 pp. 159-169 (1980).
3. A. P. Reeves, "Parallel Pascal: An Extended Pascal for Parallel Computers," *Journal of Parallel and Distributed Computing* 1 pp. 64-80 (1984).
4. A. P. Reeves, "The Anatomy of VLSI Binary Array Processors," in *Languages and Architectures for Image Processing*, ed. M. J. B. Duff and S. Levialdi, Academic Press (1981).
5. R. W. Gostick, "Software and Algorithms for the Distributed-Array Processor," *ICL Technical Journal*, pp. 116-135 (May 1979).
6. NCR Corporation, *Geometric Arithmetic Parallel Processor*, NCR, Dayton, Ohio (1984).
7. A. P. Reeves and C. H. Moura, "Data Manipulations on the Massively Parallel Processor," *Proceedings of the Nineteenth Hawaii International Conference on System Sciences*, pp. 222-229 (January, 1986).
8. P. J. Burt, "Fast Filter Transforms for Image Processing," *Computer Graphics and Image Processing* 16 pp. 20-51 (1981).